电气信息工程丛书

Linux PowerPC 详解——核心篇

王 齐 编著



机 械 工 业 出 版 社

本书分 8 章，第 1 章讲述 Linux PowerPC 的组成；第 2～4 章讲述了有关 PowerPC 处理器的基础知识，包括指令集、寄存器、内存体系结构等；第 5～8 章讲述 Linux 系统在 PowerPC 处理器中的运行，包括进程调度、中断处理、内存管理和初始化。

本书奉献给所有热爱 Linux 及 PowerPC 处理器的读者。

图书在版编目（CIP）数据

Linux PowerPC 详解——核心篇/王齐编著.—北京：机械工业出版社，2007.9

（电气信息工程丛书）

ISBN 978-7-111-22425-9

Ⅰ.L…　Ⅱ.王…　Ⅲ.Linux 操作系统　Ⅳ.TP316.89

中国版本图书馆 CIP 数据核字(2007)第 148480 号

机械工业出版社(北京市百万庄大街 22 号　邮政编码　100037)
策　　划：时　静
责任编辑：车　忱
责任印制：李　妍

保定市中画美凯印刷有限公司印刷

2008 年 1 月第 1 版·第 1 次印刷
184mm×260mm·26.5 印张·657 千字
0001—4000 册
标准书号：ISBN 978-7-111-22425-9
定价：40.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
销售服务热线电话：(010)68326294
购书热线电话：(010)88379639　88379641　88379643
编辑热线电话：(010)88379739

前　　言

许久以前，我有编写此书的想法。只是当时忙于各样的工程，没有时间。后来发现所谓没有时间不过是一个美丽的谎言罢了。两年前的一天，我重拾昔日的念头，坚持下来，就有了这本书。

我最早于1997年接触UNIX，并断断续续做了一些与UNIX有关的项目，至今难忘当时分析OSF 4.0D文件系统的日日夜夜，也难忘曾经给予我帮助的朋友们。我与Linux的第一次接触是在1998年，当时对这个操作系统嗤之以鼻。UNIX主要分为两个流派：BSD和SYS V。我是BSD的积极拥护者，认为Linux不过是几个学生的玩具罢了。在2000年末，我开始真正学习并使用Linux系统。当时需要做一个与Intel StrongARM相关的项目，Linux被事先选定。我虽不喜欢Linux，但也无奈，就此开始了对Linux的研究。

那时研究开发Linux并不是一件容易的事情，采用的基本方法是从Internet上搜集只言片语，再与Linux源代码对照。采用此方法的好处是能够体会到梅花香自苦寒来的道理。时过境迁，Linux与当年的版本不可同日而语。现在国内外有许多基于奔腾处理器的Linux书籍，但是鲜有基于PowerPC处理器的Linux书籍。我愿先行一步，以抛砖引玉。

笔者写作此书的原意是想用朴素的语言描述一款处理器和在其上运行的操作系统，最终选择了PowerPC作为处理器，Linux作为操作系统。这一选择并非偶然，也并非因为笔者对PowerPC处理器和Linux更为熟悉。在激烈的竞争和日新月异的技术更新过程中，各个流派的操作系统与各种类型的处理器，其体系结构趋同。读者真正掌握了一种操作系统和一款处理器结构后，理解其他操作系统和处理器是水到渠成的。PowerPC与其他处理器相比，其体系结构相对开放。Linux更是公开了所有源代码。两个开放的系统最终走到一起，是自然的选择。

本书分为两大部分，由8章组成。第1章至第4章是第一部分，第1章概括了Linux系统的现状；第2、3章重点描述了PowerPC的一些基础知识，包括PowerPC的指令集、寄存器和内存体系结构；第4章讲述了一款PowerPC处理器实例。第一部分的内容是学习第二部分内容的基础。第二部分由第5章至第8章组成，第5章至第7章讲述了Linux PowerPC的三大模块，进程管理与调度、中断与异常的处理及内存管理机制；第8章介绍Linux PowerPC的初始化。第二部分内容互相交织，可能读者在通读本书后，才能够理解这些内容。本书的有些内容与具体实现细节相关，而且这些细节并没有完全在本书中讲述清楚，这要求读者必须对照Linux源代码，深入了解这些细节。

凡过分拘泥细节的书籍必非入门书。本书也不是Linux系统的入门书籍。笔者不是不想写入门书籍，而是没有这个能力。入门者对新事物往往没有分辨能力，于是非大师不写入门书籍。恐怕只有大师级的人物才能够合理地避重就轻，写出适合初学者的书籍。

本书不是入门书，又是什么？对于这个问题，我在两年前有一个很清晰的答案。著书必求传诵。在写书的过程中，我陆续复习着一些经典之作，有些书看了不下十几遍，也由此明白能让读者反复阅读，而且每次阅读都有新收获的，就是经典书籍。即便这些书籍被束之高阁，也不能遮挡它们的光芒。

明白何为经典的道理后，我仍在坚持写完这本书，也曾有几次想过放弃，于是在桌旁反

复写着“也许世界上最容易的事情莫过于放弃”这样的文字。在“坚持，不放弃”的过程中，我依然以为这本书不会成为经典也至少可以会对人有益，倘若真的如此也足以令我欣慰。

在今年的某一天，我完成了本书的主体，并在剩余时间里，反复修改本书，包括文字的调整，错误的修正等。发现 Bug 的数目与修改时间成正比，似乎书中的 Bug 无穷无尽，而唯一能够消除这些 Bug 的方法就是停止这些修改，将这本书奉献给大家，这也是本书被催熟的原因。

完稿之余，不胜恐慌，本书虽经李阳，吴晓川及北京的其他同事认真讨论和细致校对，我仍日夜惴惴，害怕本书带给读者更多的是误导。恐慌已极，我明白了一个道理，凡书必有错误，有人用心去读，必会引发一些争议，写书无法苛求永不犯错，重要的是及时改正。

如果读者对此书有批评，我将在 sailing.w@gmail.com 中接受你们的批评；如果有争议，我将在 www.zh-kernel.org 等待你们的争议，读者可以订阅这个网站的 Linux-kernel 邮件，然后 E-mail 给邮件列表中的所有成员。

最后感谢的是我新婚的妻子范淑琴女士。我们相爱十余载，去年结婚，正是本书写作的关键时期。那时我送给她的最大礼物莫过于心中默念了几遍“执子之手，与子偕老”。相知十年，聚少离多，我鲜有机会送她一份像样的礼物，谨将此书奉献给她。没有她，将不会有这本书。

作　者

目　　录

第1章 什么是嵌入式 Linux

许多工程师对嵌入式 Linux 这一称呼感到陌生,他们认为产生这一称呼的主要原因是这些 Linux 系统运行在“所谓的”嵌入式处理器之上,于是产生了基于 ARM 处理器的嵌入式 Linux 系统,基于 MIPS 处理器的嵌入式 Linux 系统和基于 PowerPC 处理器的嵌入式 Linux 系统。

对嵌入式 Linux 的这种解释似乎十分合理,但是也许还会有人继续深究,提出另外一个问题,究竟什么叫做嵌入式处理器?在处理器刚刚诞生时,没有人称哪一类处理器为嵌入式;在个人 PC 诞生前夜,也没有哪一类处理器被冠以嵌入式的名号。至少当年 Motorola(现在的 Freescale 半导体)的 68K 系列处理器与 Intel 的 8086 处理器争夺个人 PC 处理器候选资格时,没有人将 68K 系列的处理器归类为嵌入式处理器,虽然目前 68K 系列的处理器几乎被认为是嵌入式处理器中的典范与鼻祖。

嵌入式处理器这一称呼始于 20 世纪 90 年代中期,那时摩尔定律一次又一次地得到验证,Intel 与 Microsoft 一起迎接了一次又一次的辉煌。其他的处理器,及运行在其上的操作系统被 Wintel 笼罩,显得愈发式微。

在这一大背景之下,嵌入式这个概念从各个主流处理器厂商的市场人员中产生。不仅各种各样的处理器被冠以嵌入式的头衔,连许多 DSP 芯片也被称为嵌入式 DSP。至今,嵌入式的概念也已经十分模糊了,似乎嵌入式无处不在。

但是经过稍微仔细的分析后,我们可以发现,除了在服务器中使用的处理器和 Pentium 处理器外,其他的所有处理器都被称为嵌入式处理器。许多人认为,工业控制上采用的 8/16 位单片机,手机上使用的 ARM 处理器,在电信领域得到广泛应用的 PowerPC 处理器,甚至 DSP 处理器都是属于嵌入式的。起码这些处理器都符合嵌入式处理器的典型定义,易裁剪,面向应用。

1.1 嵌入式 Linux 概述

所谓嵌入式 Linux 不过是标准 Linux 发布包(Linux Distribution)在不同种类处理器上的应用。这些应用包括 Linux Distribution for PowerPC(基于 PowerPC 处理器的 Linux 系统),基于 ARM 处理器的 Linux 系统,基于奔腾处理器的 Linux 系统和基于 MIPS 处理器的 Linux 系统。这些 Linux 的内核都来自 www.kernel.org。

Linux 系统中,还有一种被称为 uClinux 的一个 Linux 系统分支,uClinux 源于 Linux 系统,但是 uCLinux 只支持没有 MMU(Memory Management Unit)的处理器,如 Freescale 半导体的某些 Corefire 处理器和 Samsung 公司的一些低端 ARM 处理器。

最初 uClinux 由 Lineo 公司开发并维护,后来 Lineo 公司被一家软件公司 Metrowerks 收购,Metrowerks 又被 Motorola 收购,并随着 Freescale 与 Motorola 分离而分离。在这一系列的分离收购过程中,也伴随着一些优秀工程师的离去。这些离去者依然坚守着 uClinux 阵营,因

此我们现在仍然可以在 www.uClinux.org 中看到有关 uClinux 的最新更新与发展。目前 Linux 2.6 内核已经可以支持没有 MMU 的处理器,uClinux 这一分支也许很快就会消失。

大多数使用过 Linux 系统的用户是从基于奔腾处理器的 Linux 系统开始。Linux 系统中的绝大部分源代码都是与体系结构无关的,因此运行在其他处理器上的 Linux 系统和在奔腾处理器上运行的 Linux 并没有本质上的不同。程序员如果掌握了基于奔腾处理器的 Linux 系统,对于学习基于其他处理器的 Linux 系统十分有帮助。

任何一个 Linux 系统都是运行在一个处理器系统中的。一个典型的处理器系统一般包含一个或者多个处理器,存放数据或者程序的内部存储器,保存程序或者数据的非易失的外部存储器,和一些外部设备,如以太网口和串行口。

一般来说,在工业控制领域和电信领域中使用的处理器系统所具有的硬件资源与奔腾处理器系统相比较少。在多数情况下,这些处理器系统中并没有硬盘,因此也不存在对换区。运行在这类处理器系统中的 Linux 系统只有相对较小的内部和外部存储器。因此系统设计者需要对这类 Linux 系统进行裁剪,但无论如何裁剪,仍五脏俱全。一个典型的 Linux 系统由以下内容组成:

- 引导程序(Boot Loader)。常用的引导程序有 Lilo,Grub,Redboot,Yaboot 和 U-Boot,其中 Lilo 和 Grub 多用在基于奔腾处理器的 Linux 系统中;Redboot 主要用在基于 ARM 处理器的 Linux 系统中;而 Yaboot 和 U-Boot 分别用在 IBM 的 PowerPC 处理器和 Freescale 的 PowerPC 处理器中。
- Linux 内核映像。Linux 内核映像中主要包括进程调度,内存管理,中断/异常处理系统,网络系统,文件系统,外部设备驱动系统等。用户可以根据需要对 Linux 内核的这些子系统进行裁剪。一般来说,Linux 内核映像被压缩存储在处理器系统中。
- 根文件系统映像。大多数应用都需要一个根文件系统用来保存用户程序,此外在根文件系统中还存放着一些与系统资源有关的文件。

与基于奔腾处理器的 Linux 系统相比,基于其他处理器的 Linux 系统的基本模块并没有本质的不同。只是在基于其他处理器的 Linux 系统中,对 Linux 内核引导时需要根据体系结构的特点进行初始化。有些与处理器系统相关的资源如 MMU、中断、系统调用等,必须根据自身系统的特点进行处理。

Linux 系统的开发维护者将与处理器体系结构有关的源代码全部放入 ./arch 目录,并与其他代码相对独立。在 ./arch 目录中有许多子目录,如 alpha,arm,PowerPC,mips 等等。这些子目录分别与基于 Alpha 处理器、基于 ARM 处理器、基于 PowerPC 和基于 MIPS 处理器的 Linux 系统对应。本书主要介绍基于 PowerPC 处理器的 Linux 系统,并将此简称为 Linux PowerPC。

1.2 什么是 Linux BSP

BSP 是 Board Support Package 的缩写。BSP 这个概念首先出现在 Windriver 的 Vxworks 系统中,BSP 用于描述处理器硬件信息并将这些信息传递给操作系统,在 BSP 中还包含一些基本的设备驱动程序。通过修改 BSP,系统程序员可以使 Vxworks 系统运行在不同种类的处理器系统中。程序员在编写 Vxworks 系统的 BSP 时一般要遵循某种处理器系统的 BSP 格

式,大多数 BSP 的开发工作会在某一个成型的 BSP 的基础上进行。

一般来说,BSP 是针对某个特定的处理器的,Windriver 会提供一些针对某些处理器系统的 BSP,如 BSP for MPC8272ADS 板,BSP for MPC8560ADS 板,BSP for MPC8555CDS 板等等,MPC8272ADS,MPC8560ADS 板和 MPC8555CDS 板是 Freescale 针对不同种类处理器开发的评估板。

然而,在许多书籍和文章中还出现了 Linux BSP 这个概念。这个概念在诞生之时就引发了诸多争执。许多 Linux 系统的开发者至今也不承认有 Linux BSP 的存在,他们认为所谓的 Linux BSP 只是借用了 Vxworks 系统中 BSP 的概念,把 Linux 系统中与处理器相关的代码进行抽象,并将 Linux 系统的初始化代码(head.S 文件)、体系结构初始化代码(setup _ arch 函数)与一些设备驱动程序的开发工作统称为 Linux BSP 的开发。

与 Vxworks 系统不同,Linux 系统中所谓的 BSP 与 Linux 发布包之间并不容易分割,Linux 系统也并没有专门的接口用来开发所谓的 Linux BSP。这也是许多 Linux 系统工程师不承认有 Linux BSP 的主要原因。

对不同的 PowerPC 处理器,Linux PowerPC 的实现大体相同。Linux PowerPC 将与各类 PowerPC 处理器有关的代码放在 ./arch/powerpc/platforms 目录中。如与 MPC8555CDS 板有关的代码存放在 ./arch/powerpc/platforms/85xx 目录的 mpc85xx _ cds.c 和 mpc85xx _ cds.h 文件中,与 MPC8272ADS 板有关的代码存放在 ./arch/powerpc/platforms/85xx 目录的 mpc82xx _ ads.c 文件中。此外,在 ./arch/powerpc 的其他子目录中还包含一些与处理器内核有关的一些代码。这些代码共同组成了所谓的 MPC8555CDS 板的 Linux BSP。

1.3 Linux 系统的相关标准

Linux 的繁荣源于商业和技术的需求。许多人认为一个源代码公开而且免费的操作系统可以节约人力及物力成本,何况这个免费源代码的操作系统还有许多人在维护和发展。

还有许多人认为源代码开放的操作系统会为整个项目带来更高的可靠性、可维护性,并可以提高效率,至少他们在发现自己某些设备驱动程序的缺陷或者效率低下时,可以参照 Linux 系统中的源代码进行改进,也许这些源代码会帮助程序员查找到一些蛛丝马迹。但是这里需要提醒程序员注意,在 Linux 系统中进行开发时,需要遵循一些基本原则。

1.3.1 GPL 与 LGPL

GPL 是 General Public License 的缩写,而 LGPL 是 GNU Lesser General Public License 的缩写,其中 GNU 是 GNU's not UNIX 的递归缩写。

GPL 与 Copyleft(非赢利版权)组成了自由软件的发展基石。GPL 和 LGPL 的原文在以下网址:

http://www.gnu.org/copyleft/gpl.html

http://www.gnu.org/copyleft/lesser.html

http://www.linux.org.tw/CLDP/OLD/doc/GPL.html

http://www.linux.org.tw/CLDP/OLD/doc/LGPL.html

GPL 赋予每个使用者使用、修改和复制 Linux 源代码的权利的同时,也传递了使用基于

GPL 软件时的义务。创建 GPL 的目的是反对所谓的版权，但 GPL 同时依靠版权法和相关的游戏规则来保证自由软件的发布与共享。GPL 是特别强调版权的。GPL 首先要求在其下发布的程序要有“版权说明”；之后是著名的“自由软件声明”；然后声明本程序的作者对此程序不做担保；最后通知大家，您已经随这份程序收到了 GPL。

GPL 赋予使用者复制、传播和修改自由软件的权利，并向使用者免费提供自由软件的源程序。为此，您只需要做两件事情，一是标明版权声明和无担保声明，二是继续传递 GPL 给接收者。GPL 要求使用者原封不动地将 GPL 连同基于 GPL 的源代码一并进行复制、传播；GPL 严格限定自由软件转化为“专有权(proprietary)”的可能性，并对自由软件可能受到的“专有权”威胁进行了约定，即任何基于 GPL“专有权”的软件必须保障每个人可自由使用这些代码而无需授权。

简单地说，GPL 的主要目的是首先声称自己是有“专有权”特性的，这个“专有权”需要保证基于 GPL 的软件不能成为具有“专有权”特性的软件。GPL 是一个利用版权法强调的“专有权”对版权法所保护的“专有权”进行打击的协议。而 GNU/Linux 的成功也证明了 GPL 的成功。

LGPL 是针对函数库的条款。不同于 GPL，它的感染性并不十分强。LGPL 最初被称为 GNU library General Public License。LGPL 是针对函数库而设计的授权条款，并弱化了 GPL 所追求的软件自由。LGPL 采取了一种务实的推广态度，并提供了比 GPL 更加商业友善的共享方式。LGPL 首先要求划分“基于函数库”和“使用函数库”的区别。基于函数库是指使用基于 LGPL 函数库的程序，此类程序将被 LGPL 感染成为新的 LGPL 函数库。而“使用函数库”是指简单利用或微量取用 LGPL 函数库的程序，如使用这些库内定义的头文件或内联函数，这些程序不被 LGPL 所感染。

以上规定仅是有关 LGPL 的大概解释，详细的分别需视情况而定。

Linux 系统是也是基于 GPL 的。GNU 使 Linux 一夜之间获得了可以和任何商业 UNIX 抗衡的所有软件，如 Tool-Chain，图形界面及各种各样的应用程序，也正是因为 GNU 对 Linux 系统的巨大推动作用，因此 GNU 软件的创始人 Richard Stallman 一再建议大家“要把 Linux 系统称为 GNU/Linux 系统”。

1.3.2 有关 Linux 系统的规范

十年前有许多 UNIX 流派，如 OSF，Saloris，SCO，HP-UX，AIX 等。当时几乎没有人认为 Windows(当时 Win95 刚刚问世不久)会大举进入服务器市场，当然这个结论到现在也是正确的。也没有太多人过于在意 Linux 这个没有任何担保的操作系统。当时摆在 UNIX 程序员面前的一个巨大难题是，UNIX 程序员必须要针对不同种类的 UNIX 分别编写出不同版本的程序。当时不同 UNIX 系统间的应用程序在源代码级都不能做到互相兼容，程序员编写基于 OSF 或者 Saloris 系统的应用程序时，或者需要为不同的操作系统单独书写程序，或者需要在这些程序中进行某些兼容性处理，以使得这些程序可以在不同的 UNIX 系统中编译。

UNIX 系统产生的目的是为用户提供一个标准化和开放性操作系统，但是后来却逐步演变成新的专属操作系统，这也制造了一场漫长持久与猛烈的“UNIX 大战”，当时不同版本的 UNIX 可谓是百花齐放，百家争鸣。这场战争至今尚未结束，只是参加“UNIX 大战”的斗士们突然发现在旷日持久的战斗期间，Microsoft 大踏步地奔向银行并很快成为首富，Linux 系统也

利用了他们的战争衍生物 POSIX 迅速成长,迅速普及。这一切使得在技术层面上有关 Linux 系统和 UNIX 系统孰优孰劣的讨论变得没有价值。至今最令人惊奇的不是 Linux 系统的发展壮大,而是还有些 UNIX 系统依旧活着。在可以预料的将来,Linux 必然会进一步壮大,可以阻止这一进程的只有 Linux 自己。

Linux 系统的发布者吸取了 UNIX 的发展教训,及时地公布了一系列的标准,并要求系统程序员在书写程序时严格按照这些标准执行。

(1) LSB 标准 。LSB 标准(Linux Standard Base)的设计目标是为了提高 Linux 系统发布版本之间的兼容性。LSB 标准由 FSG(Free Standard Group)开发和维护,LSB 标准由与体系结构相关的设计文档、测试软件包、二进制接口标准 ABI(application binary interface)、各种实施细节等一系列文档组成。

LSB 标准的设计目标是使应用程序在任何 Linux 发布版上运行。FSG 董事会成员 Dirk Hohndel 预测,尽管 LSB 标准的目标不是建立一种单一的 Linux 操作系统,但它将提供一个环境,在这个环境中,所有基于 LSB 标准的各种 Linux 发布版本可以相互兼容,这使得使用 Linux 系统的用户可以在所有的 Linux 系统中使用支持 LSB 标准的应用软件。

有关 LSB 标准的详细资料可参见以下网址:

http://lsb.freestandards.org/~mats/lsbspec-3.1rc1/specs.php

(2) POSIX(Portable Operating System Interface)标准。POSIX 标准由 IEEE 和 ISO/IEC 共同开发。POSIX 标准的提出是为了提高 UNIX 环境下应用程序的可移植性。然而 POSIX 标准并不局限于 UNIX,许多其他的操作系统,如 Windows,Vxworks 也支持这个标准。

POSIX 标准规定了各个操作系统需要共同实现的接口标准,从而便于应用程序在源代码一级上的移植。

1991~1993 年,Linux 系统刚刚起步,此时 POSIX 标准正处在最后的定稿阶段。因此 Linux 系统自然而然地选择了 POSIX 标准。POSIX 标准为 Linux 系统提供了极为重要的帮助,使得 Linux 系统中的应用程序能够与绝大多数 UNIX 系统的应用程序兼容。

最初的 Linux 内核代码(0.01 版、0.11 版)就已经为 Linux 与 POSIX 标准的兼容做好了准备,Linux 在发展初期就想实现与 POSIX 的兼容。从 Linux 的发展进程也可以看出, Linux 的成长一直有 POSIX 标准的辅佐,没有 POSIX 的指导,就不会有 Linux 的今天。

目前 POSIX 标准已经被国际标准化组织 ISO(International Standard Organization)所接受,被命名为 ISO/IEC 9945-1:1990 标准。读者可以在以下网址中,进行免费注册,然后阅读此规范:

http://www.unix.org/version3/online.html

(3) OSDL(Open Source Development Lab)标准系列。OSDL 支持的标准中最为有名的就是 CGL(Carrier Grade Linux),CGL 致力于制定符合电信运营的 Linux。CGL 承诺遵循 LSB 标准。此外 OSDL 还资助了 DCL(Data Center Linux)和 DTL(Desktop Linux)标准。有关 OSDL 的标准请参见 http://www.osdl.org。

1.4 Linux 系统的主要发布版本

由上文可知,Linux 系统由 Boot Loader 程序、内核程序及许多应用程序组成。虽然 Linux

系统中的所有源代码都是基于 GPL 或者 LGPL 的，但是将所有的这些应用程序和 Linu 内核组织在一起，并进行测试与维护，最后形成一个完整 Linux 发布包并没有想象中那么容易，尽管许多读者有编译并构造 Linux 发布包的经验。

互联网上有一个标准文档 LFS(Linux From Scratch)，介绍如何使用散布在各个网站的源代码构建一个 Linux 发布版本的过程(LFS 的网站为 http://www.linuxfromscrach.org)。但是将这些 Linux 发布版本，最后封装成为产品，发布到最终用户手里，还要需要许多具体和细致的工作。目前，较为常用的 Linux 发布包有以下几种。

(1) Redhat。RedHat Linux 是目前世界上使用最多的 Linux 发布包。它具备最好的图形界面，无论是安装、配置还是使用都十分方便，而且运行稳定。因此不论是新手还是老玩家都对它有很高的评价。RedHat Linux 9.0 以上的版本更名为 Fedora。

(2) Debian。Debian 是 Linux 发行版当中最自由的一种，Debian 不属于任何商业公司。与 Redhat 相比，Debian 的安装有些难度。不过，使用 Debian 可以实现应用软件平滑升级，Debian 还具有最为丰富的软件包。Debian 的软件包管理程序较为强大，有许多使用 Linux 系统的专业人士喜欢 Debian。

(3) SUSE。SUSE 是德国最富盛名的 Linux 发布版，SUSE 继承了德国人一贯严谨的作风。在欧洲，SUSE 是 Linux 企业级版本的首选。2003 年 SUSE 被 Novell 收购。

(4) Gentoo。Gentoo 是唯一基于源代码的 Linux 发布版本。Gentoo 在安装时可以选择预先编译好的软件包，但是大部分使用 Gentoo 的用户都选择自己手工编译所有与 Linux 系统相关的代码。由于与 Linux 系统相关的代码实在太多，因此编译完软件需要消耗大量的时间。如果编译所有与 Linux 系统相关的软件，并安装 KDE 桌面系统等比较大的软件包，有可能需要几天时间才能将 Gentoo 安装完。

(5) Yellow Dog。Yellow Dog Linux 是 Linux PowerPC 的主要参与者之一。Yellow Dog Linux 与 Red Hat Linux 较为类似。与 Red Hat Linux 相比，Yellow Dog 更加偏重于对 Linux PowerPC 的支持。

(6) Ubuntu。Ubuntu 基于 Debian，使用 apt-get 命令进行软件更新。但是 Ubantu 与 Debian 略有不同，Ubuntu 每 6 个月发布一次，并为每一个版本提供 18 个月的技术支持。2006 年以来，Ubuntu 取得了长足的进步，并逐渐为用户接受。目前 Ubuntu 已经成为一种主流的 Linux 发布包。

1.5 Linux 系统的组成

1991 年 4 月，Linus Torvalds，即 Linux 的创始人，开始了更新 Minix 的工作。此时他写出了第一封 E-Mail，请求来自互联网上的自愿者协助开发 Linux 系统。

1991 年 9 月 Linux V0.01 面世，1994 年三月发布 Linux V1.0。V1.0 是一个真正意义上可以使用的 Linux 系统，当时全球共有 10 万个用户，中国第一批 Linux 系统的爱好者就是在这时开始接触 Linux 的。此后的 Linux 系统历经了一系列主要的里程碑：V1.2(1995 年 3 月)，V2.0(1996 年 6 月)，V2.2(1999 年 1 月)，V2.4(2001 年 1 月)，V2.6(2005 年 11 月)。至今 Linux 已发展至 V2.6。

Linux 可以在不同体系结构的处理器上运行，包括 Alpha，ARM，Pentium，PowerPC 处理器

等等。Linux 系统中大多数源代码使用 C 语言编写而与硬件体系结构无关，而部分源代码使用相应的汇编语言编写并与硬件体系结构相关。本书将重点介绍与 PowerPC 体系结构相关的 Linux 系统，即 Linux PowerPC。

与其他体系结构的 Linux 系统类似，Linux PowerPC 主要由五个子系统组成：进程调度、内存管理、文件系统、网络接口、进程间通信。

(1) 进程调度。Linux 系统作为多用户操作系统，支持多进程与多线程，进程调度模块负责进程管理，并管理进程对系统资源的访问。进程调度子系统的主要作用是选择合适的进程在当前处理器系统中运行，并对未获得处理器资源的进程进行管理。Linux 系统使用基于优先级的进程调度算法。

(2) 内存管理。内存管理是进程通过软硬件协作来进行内存访问的一种机制。它可以为进程分配、释放内存，并维护系统中内存的使用状态。在 Linux 系统中，内存管理子系统较为复杂，内存管理子系统也是整个 Linux 系统的核心。如果进程调度子系统是 Linux 的心脏，那么内存管理子系统就是 Linux 的血液。

(3) 文件系统。在 Linux 中文件是一个线性的数据流，可以是存放在外部存储器中的数据，也可以用来表示某个外设。文件系统主要用来对文件进行管理。文件系统隐藏了具体设备的硬件细节，并为所有的外部设备提供了统一的接口。文件系统由逻辑文件系统(如 ext2, ufs, romfs, jffs2 等)和一些设备驱动程序组成。

(4) 网络接口。网络接口用来支持各种网络结构和各种网络硬件。网络接口可分为协议栈和网络驱动程序。网络协议部分用来实现每一种可能的网络传输协议。网络设备驱动程序负责与硬件设备通信。

(5) 进程间通信(IPC)，支持进程间各种通信机制。

Linux 的五大子系统之间相互依赖。Linux 系统使用了许多源代码用来实现其五大子系统。Linux 2.6.20 的源代码由两万多个文件组成，分布在一千多个目录中，大约有几百万行代码，共占有两百多兆的数据空间。Linux 的源代码基本结构如图 1-1 所示。

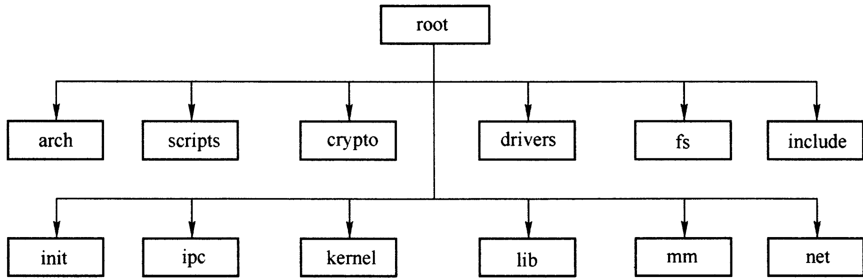


图 1-1 Linux 源代码结构图

(1) root 目录。在 root 目录中最重要的文件是 Makefile。在这个文件中包含了许多与 Linux 内核配置有关的信息，如 Linux 内核的版本号，Tool-Chain 的选择，编译连接 Linux 内核使用的参数，及各种对 Linux 系统源代码进行操作的命令。

(2) arch 目录。在 arch 目录中包含与体系结构有关的子目录与文件。Linux PowerPC 开发者所涉及的文件基本上都在 arch/powerpc 目录中。

(3) scripts 目录。在 scripts 目录中存放着许多对核心进行配置的脚本文件。

(4) crypto 目录。在 crypto 目录中包含了许多常见的密码算法,但是这些算法基本上都没有针对处理器进行优化。

(5) drivers 目录。在 drivers 目录中含有各种各样的设备驱动程序,包括字符型,块型和网络设备驱动程序。

(6) fs 目录。在 fs 目录中包含了 Linux 系统所支持的各种文件系统类型。常用的文件系统有 ext3,nfs,jffs2 和 vfs 类型。

(7) include 目录。在 include 目录中包含了 Linux 系统所用的头文件,此外在该目录中还含有与体系结构有关的目录。如与 PowerPC 处理器有关的目录 ./include/asm-powerpc。

(8) init 目录。在 init 目录中存放着与 Linux 内核相关的启动代码。其中 main.c 中的 start_kernel 函数是 Linux 内核的入口函数。

(9) ipc 目录。在 ipc 目录中存放着与 Linux 进程间通信相关的代码。

(10) kernel 目录。在 kernel 目录中含有许多与 Linux 进程调度子系统相关的源代码。与体系结构相关的部分 Linux 核心代码存放在 ./arch 目录中,如 Linux PowerPC 使用的部分核心代码存放在 ./arch/powerpc/kernel 目录中。

(11) lib 目录。在 lib 目录中存放着 Linux 内核所用的库文件。

(12) mm 目录。包含所有 Linux 内存管理与体系结构无关的源代码。与体系结构相关的部分 Linux 核心代码存放在 ./arch 目录中,如 Linux PowerPC 使用的部分与内存管理相关的源代码存放在 ./arch/powerpc/mm 目录中。

(13) net 目录。在 net 目录中存放着有关网络协议栈的源代码。

Linux 作为一个开放源代码的操作系统,其提供的源代码为广大的爱好者提供了一个深入钻研其实现细节的机会。然而面对如此海量的 Linux 源代码,即使是最有经验的系统程序员也会觉得困惑,感到难以阅读。

目前有许多工具可以对源代码的结构进行分析,如运行在 Linux 平台中的 Gvim,Emacs 文本编辑器,运行在 Windows 平台的 Source Insight 等其他工具。在互联网,还有人提供了一些专门用于阅读 Linux 源代码的网站,如 http://lxr.linux.no,便于程序员阅读 Linux 系统源代码。

1.6 什么是 Linux PowerPC

本书将基于 PowerPC 处理器的 Linux 系统,简称为 Linux PowerPC。目前为止,Linux 系统对两种处理器系统的支持最为全面。一个是 Linux Pentium,另外一个就是 Linux PowerPC。

Linux Pentium 是 Linux 用户最经常使用的操作系统,大多数 Linux 工程师是从 Linux Pentium 开始认识整个 Linux 系统的,许多人认为的 Linux 系统就是指 Linux Pentium。对 Linux 源代码进行配置时,如果用户不将 ARCH 参数进行赋值,将默认使用 Linux Pentium 作为编译对象,造成这种现象的原因是不言而喻的。

而 Linux PowerPC 的兴起主要源于 IBM 的努力,在 IBM 的 iSeries,pSeries 及许多种类服务器上都将 Linux 系统作为选择提供给用户,同时 IBM 对 Linux 系统中各个子系统都有突出的贡献,相信所有熟悉 Linux 系统开发的读者,都应该知道 IBM 的 Linux 论坛的 URL:

www.ibm.com/developerworks/cn/linux

在这里，你几乎可以找到关于 Linux 系统从内核到应用的所有入门知识；在这里，许多经典的英文文章都被翻译为汉语；在这里，IBM 的工程师和许多其他公司的工程师无私地向外界传递着有关 Linux 的各种知识。

同时，在 IBM 数以万计的工程师们直接或者间接地从事着与 Linux 系统有关的这种应用开发。这一切使得 Linux PowerPC 成为除 Linux Pentium 之外，最为成熟，使用最为广泛，也最易商业化的 Linux 系统。

Linux 系统的发展一刻也没有停息过，Linux 的各个子系统不断更新。即使都是在 Linux 2.6 版本中，2.6.10 与 2.6.20 相比，也有许多子系统发生了显著的变化。

在基于 2.6.10 的 Linux PPC 中，我们可以使用 ARCH = ppc 或者 ARCH = ppc64 对 Linux PowerPC 内核的 32 位和 64 位处理器进行配置，而到了 2.6.20 内核中我们可以使用 ARCH = ppc 或者 ARCH = powerpc 对 Linux PowerPC 内核进行配置。

在不久的将来 ARCH = ppc 将会被逐步淘汰，将会使用 ARCH = powerpc 对所有的 Linux PowerPC 支持的处理器系统进行配置。与之相关的 ./arch/ppc，./arch/ppc64，./include/asm-ppc，./include/asm-ppc64 目录也将合并到 ./arch/powerpc 和 ./include/asm-powerpc 目录中。这也是本书将基于 PowerPC 处理器的 Linux 系统，简称为 Linux PowerPC 而不是 Linux PPC 的主要原因。

与 Linux PPC 相比，Linux PowerPC 做出了许多重大的改动。

(1) 将 ppc 与 ppc64 目录进行合并，并使用 powerpc 目录对所有基于 PowerPC 处理器的 Linux 系统进行支持。

(2) 重新编写了 Linux 中断处理子系统，将之前 Linux PPC 对 OpenPIC 构架进行支持源代码更换为 Linux PowerPC 中 MPIC 构架。

(3) 全面使用 Open Firmware 结构对 Linux PowerPC 进行引导。

(4) 全面使用 LMB(Logic Memory Block)结构对内存进行管理。

(5) 重新调整 Linux PowerPC 的源代码结构，并增添了许多有利于 Linux PowerPC 层次化的数据结构，以便系统用户为 Linux PowerPC 增加新的处理器平台。

(6) 还有许多在 Linux PowerPC 中开始使用，后来提交到整个 Linux 系统中的模块。

Linux PowerPC 对整个 Linux 系统的贡献是不可磨灭的。目前，PowerPC + Linux 已经成为一个稳定的、久经考验的系统解决方案。Linux PowerPC 也正逐渐成为 PowerPC 处理器平台的首要选择。

在介绍 Linux PowerPC 之前，本书将较为详细地介绍 PowerPC 处理器的构架。如果读者已经十分熟悉 PowerPC 处理器的构架，可以略过第 2 章到第 4 章的内容。但是我仍然建议读者仔细阅读这些内容，毕竟这些内容并不是全部都可以在 PowerPC 处理器的参考手册中查找到的。

第2章　PowerPC处理器的指令集与寄存器

指令集与寄存器是PowerPC处理器中最基本的内容。软件工程师在开始接触PowerPC处理器时，就会学习这部分内容。然而在国内，许多使用过PowerPC处理器的软件工程师并不能写出漂亮的PowerPC汇编程序，多数软件工程师写出来的PowerPC汇编程序的执行效率比C程序的执行效率还要低得多。这是因为大多数软件工程师并没有全面地了解PowerPC处理器的指令集，没有深入理解PowerPC处理器的指令执行过程。

本章将以PowerPC E500内核为例，讲述PowerPC处理器的指令集与寄存器，此外还对指令在E500内核中的执行过程、乱序执行、ABI(Application Binary Interface)等知识简单地进行介绍。

2.1　PowerPC处理器概述

PowerPC处理器是Freescale、IBM和苹果电脑的合作结晶。最初这颗处理器的目标客户是Apple的MAC机，而且该处理器还针对通信类、消费类电子等一系列广泛的应用。

自从PowerPC处理器诞生以来，这颗处理器在其应用的各个领域一直遭受着竞争对手的猛烈攻击。苹果公司的Power Book对PowerPC处理器的弃用基本上标志了Wintel在个人PC市场的垄断，也同时宣告了PowerPC处理器在个人PC市场中一个时代的结束。在通信领域里，PowerPC处理器也遇到了来自ARM和MIPS处理器的强烈挑战。只有在高端的游戏领域，PowerPC+Cell处理器取得了新的成功。但是所有这些来自商业上的压力并不能阻挡PowerPC在技术上的领先。

最近十年里，IBM半导体部门取得了一个又一个新技术突破，铜介质、低介电绝缘电介质(low-k)、应变硅技术(Strained silicon)、绝缘体硅片(SoI, Silicon-On-Insulator)、硅锗双极集成电路(SiGe-BiCMOS)等一系列有关芯片制造的新工艺不断应用到PowerPC处理器中。多核技术也早已在PowerPC处理器中实现。PowerPC处理器的主频和系统总线带宽也在不断地提高。PowerPC处理器在所有这些高新技术的笼罩下，已注定是一颗“贵族的芯片”。

对于大多数软硬件工程师，PowerPC处理器的入门并不容易。PowerPC处理器自身的复杂性增加了软件的开发难度；PowerPC处理器过于灵活的硬件配置使得硬件工程师们疲于奔命；在设计基于PowerPC处理器的系统时硬件工程师不可避免地遇到一些高速硬件设计规范，这使得CAD工程师们做着一遍又一遍的硬件仿真与模拟；PowerPC处理器的各种内核都支持多处理器结构，而对于许多应用，用户仅仅需要一个处理器，但是在PowerPC处理器的内核文档中基本上都包含多内核才涉及的内容，这给新入行的工程师带来了许多困难。因此工程们在深入理解PowerPC处理器时不可避免地会遇到许多障碍。

即便如此，仍有许多公司毅然选择使用PowerPC处理器。因为作为一个处理器，PowerPC的优点十分突出。这些优点主要体现在PowerPC的指令集、指令执行和处理器内核的设计。

IBM 和 Freescale 的 PowerPC 处理器的最大区别在于 Freescale 的 PowerPC 处理器中包含一个 QUICC 芯片。Freescale 也将包含 QUICC 芯片的 PowerPC 称为 PowerQUICC。PowerQUICC 系列的 CPU 包含两个处理器芯片：PowerPC Core 和 CPM（Communications Processor Module）。

IBM 与 Freescale 半导体维护着各自 PowerPC 处理器的设计路线。

其中，IBM 出售的 PowerPC 处理器大致分为以下几种：

- PowerPC 600 系列，如 PowerPC 603e，604e，第一个 64 位的 PowerPC 处理器 PowerPC 620 等一系列处理器。
- PowerPC 700 系列，如 PowerPC750。PowerPC750 是世界上第一个使用铜介质的芯片。PPC750FX，PPC750GX 不仅适用于 PC 市场（Apple 机），也是高端消费类市场的主角。你可以在许多高端打印机中找到这颗芯片。
- PowerPC 900 系列。64 位处理器 PowerPC 970。这颗处理器开始批量生产的时间距离 Apple 公司弃用 PowerPC 处理器的时间较为接近。
- PowerPC 400 系列。这是一个非常被看好的 PowerPC 处理器系列。这类芯片集中体现了 PowerPC 处理器灵活配置的特性，当时这类芯片的设计目标是从小型的应用系统到大规模的巨型机上。如采用低端配置的 PPC405EP，可以适合机顶盒，小型交换机的需求；而采用高端配置的 PPC440GX，可以适合高端路由器、复杂的 3G 系统到超大规模巨型机的应用。人们正盼望 PowerPC 400 系列在市场上取得成功的时候，IBM 突然宣布将 400 系列的 PowerPC 处理器转让给 AMCC。

IBM 还有一系列用于服务器的 PowerPC 处理器芯片，只是一般用户并不能直接获得有关这些处理器的详细资料。这些 PowerPC 处理器主要用于 IBM 的服务器，近期并没有卖给其他用户的打算。从技术角度上说，这些 PowerPC 处理器的组成结构远比用于通信领域的 PowerPC 处理器复杂得多，这一类芯片是 IBM 的 PowerPC 处理器的精华，只是有关这些 PowerPC 处理器的资料并没有完全公开。

Freescale 的 PowerPC 处理器共分为两类，一类用于汽车电子，如 MPC5XX 系列，这类芯片主要用于汽车的主控系统；另一类用于通信领域的 PowerPC 处理器。用于通信领域的 PowerPC 处理器大致分为以下系列：

- 603 内核系列。如 MPC850，MPC860，MPC885 等。这类芯片目前是 Freescale 最低端的 PowerPC 处理器。这些处理器的最大优点就是价格低廉。中国第一代 PowerPC 工程师就是从 MPC860 处理器开始学习使用 PowerPC 处理器的，在这类 PowerPC 处理器中并没有包含 SDRAM 接口，用户必须使用 MPC860 中提供的 UPM（User-Programmable Machines）接口配置 SDRAM 接口。
- 603E 内核系列。包括 MPC8250，MPC8260，MPC8272 等。从 PowerPC 内核的角度看，603 内核到 603E 内核的升级并不是非常大。其中主要的升级是在存储器管理单元 MMU 上，此外 MPC82XX 系列的处理器包含了 SDRAM 控制器，可以直接与 SDRAM 芯片直接相连。
- E300 内核系列。包括 MPC8349，MPC8347，MPC8360 等。E300 系列与 603E 系列结构基本一致，在处理器内核上作的修改并不多。Freescale 的 QE 首先在 MPC8360 芯片中实现，此外基于 E300 内核的处理器支持 DDR SDRAM 接口。基于 E300 内核的处理

器可以对基于 603E 内核的处理器进行替换。

- E500 内核系列。包括 MPC8540,MPC8560,MPC8548 等。E500 内核共有两个版本，V1 和 V2。V1 版本与 V2 版本的最大区别在于 MMU。基于 E500 内核的 PowerPC 处理器是 Freescale 高端处理器的发展方向，其内核结构与 603E 内核有较大的不同。具体地说，这两类 PowerPC 内核间只有指令系统是兼容的，其他的内核组件都不相同。基于 E500 内核的芯片支持 DDR SDRAM、RapidIO 和千兆以太网等高速接口。
- G4,E600 内核系列。包括 MPC7410,MPC7447,MPC7448,MPC8641 等。这一系列芯片与 IBM 的 PowerPC 700 系列的最大不同在于 G4 系列支持 AltiVec 结构(该结构中包含一组 SIMD 指令)。这类处理器也是 Apple 公司用于 MAC 机的芯片。
- E700 系列，支持 64 位的 PowerPC 结构，正在实现中。这类芯片的内核基于 Book E 的 64 位实现，而 E500 内核是 Book E 的 32 位实现。

最近，Freescale 将 CPM 升级为 QE(QUICC Engine),QE 是 PowerQUICC 系列芯片十年以来推出的最重要的一个组件。我们可以在基于 E300 内核的 MPC8360,MPC8323 这些处理器中发现 QE 所提供的各种功能。某些基于 E500 内核的芯片也将支持 QE，如 MPC8568，MPC8572 等一系列芯片。

在某种程度上，QE 代表了 PowerQUICC 的未来，CPM 将逐步被 QE 所替代。如果 Freescale 可以将 QE 的全部源代码公开，并允许用户进行二次开发，QE 所起到的作用与一些底端的网络处理器(Network Processor,NP)类似。否则，其作用不过是对现有 CPM 模块的升级。

本书主要介绍基于 E500 内核的 PowerPC 处理器。其中，E500 内核是 PowerPC 体系结构的一个内核。基于 E500 内核的处理器一般使用在通信和工业控制领域的高端应用中，是一颗用于控制层面的处理器。

在学习 E500 内核之前，读者需要了解 Book E 内核、E500 内核和 PowerQUICC III 之间的区别。Book E 内核是 IBM 和 Freescale 共同开发的一种 PowerPC 内核。其中 Book E 内核在指令集和寄存器上与其他 PowerPC 内核完全兼容，Book E 内核描述了 32 位和 64 位 PowerPC 处理器的实现细节。

E500 内核只支持 32 位 PowerPC 处理器，该内核继承了 Book E 内核中 32 位处理器的大多数内容，但在实现的细节上略有不同。比如 E500 内核不支持浮点数据的处理。一些基于 E500 内核的处理器，如 MPC8541 使用 SPE(Signal Processing Engine)处理浮点数的运算。PQIII 系列处理器是 Freescale 基于 E500 内核的一组芯片，包括 MPC8540，MPC8560，MPC8541,MPC8548 等一系列处理器，在不远的将来，Freescale 还会推出一系列基于多个 E500 内核的 PowerPC 处理器，如 MPC8572 和 MPC8574 等。

与其他 PowerPC 体系结构(如 603E 内核)相比，E500 内核在 MMU，指令执行和中断系统做出了许多重大修改，但是在用户指令集及寄存器上与基于其他 PowerPC 内核的处理器兼容。本书将以 E500 内核为主介绍 PowerPC 体系结构，同时稍微兼顾基于 603E 的内核。E500 内核共有 V1 和 V2 两个版本，本书将对 E500 V1 版本进行介绍。

本书不会讲述有关 Book E 和 E500 内核的全部细节，对此有兴趣的读者可以从以下 URL 中下载有关 Book E 和 E500 内核的用户手册，进一步了解 Book E 内核、E500 内核及与之相关的一些实现细节：

www.freescale.com/files/32bit/doc/ref_manual/EREFRM.pdf

www.freescale.com/files/32bit/doc/ref_manual/E500CORERM.pdf

www.freescale.com/files/32bit/doc/ref_manual/E500ABIUG.pdf

如果读者已经基本掌握了 E500 内核的知识,这些手册可以用作工具书,需要时可以依此进行检索查询。但是对于初学者,这些文档可能并不合适。为此,下文会依照以上这些文档,对"如何学习 E500 内核"进行简单的叙述。

2.2 E500 内核的寄存器

E500 内核有两组寄存器,分别是用户模式寄存器(User-Level Registers,ULR)与超级用户模式寄存器(Supervisor-Level Registers,SLR)。与此相对应,E500 内核有两种运行模式,即用户模式(User Mode)和超级用户模式(Supervisor Mode)。

PowerPC 处理器在用户模式或者超级模式下运行时可以访问用户模式寄存器 ULR,而 PowerPC 处理器在超级模式下运行时可以访问超级用户模式寄存器 SLR。

用户可以通过修改 E500 内核的处理器状态寄存器 MSR(Machine State Register)的 PR 位进行用户模式与超级用户模式的切换。E500 内核也可以在进入中断、系统调用和异常时实现用户模式与超级用户模式的切换。有些相对较为简单的操作系统如 VxWorks 始终将 E500 内核运行在超级用户模式,而不进行处理器状态的切换。

Linux 系统使用 E500 内核提供的用户模式与超级用户模式。Linux 系统包含两类地址空间,分别为核心空间与用户空间。其中核心进程和在 Linux 核心中执行的用户进程运行在 Linux 核心空间中,而用户进程大多数时间运行在 Linux 用户空间中。

用户进程需要使用系统调用才能够进入 Linux 核心空间。Linux PowerPC 要求进程在 Linux 核心空间运行时,处理器的 MSR 寄存器的 PR 位为 0;进程在 Linux 用户空间运行时,处理器的 MSR 寄存器的 PR 位为 1。为此,Linux PowerPC 中专门定义了两个宏,分别用来描述处理器处于用户模式或者超级用户模式时,MSR 寄存器的值。处理器运行在用户模式时 MSR 寄存器的值为 MSR_KERNEL,而处理器运行在超级用户模式时 MSR 寄存器的值为 MSR_USER,这两个宏的定义在 ./include/asm-powerpc/reg.h 文件中。

```
#define MSR_USER (MSR_KERNEL|MSR_PR|MSR_EE)
#define MSR_KERNEL (MSR_ME|MSR_IP|MSR_RI|MSR_IR|MSR_DR)
#define MSR_ME __MASK(MSR_ME_LG) /* Machine Check Enable */
#define MSR_RI __MASK(MSR_RI_LG) /* Recoverable Exception */
#define MSR_IR __MASK(MSR_IR_LG) /* Instruction Relocate */
#define MSR_DR __MASK(MSR_DR_LG) /* Data Relocate */
#define MSR_PR __MASK(MSR_PR_LG) /* Problem State/Privilege Level */
#define MSR_EE __MASK(MSR_EE_LG) /* External Interrupt Enable */
```

由以上代码我们可以发现,当 PowerPC 处理器运行在用户模式时 MSR 寄存器的 PR 位为 1。在 Linux PowerPC 中,用户进程或者继承父进程的 MSR 寄存器的值,或者使用系统调用 execve 覆盖原来进程空间,产生新的进程,使用新进程 MSR 寄存器的值。

在 Linux PowerPC 中,一个进程 MSR 寄存器的值被存放在进程描述符中,当进程获得处

理器资源后，再将这些存放在进程描述符中的寄存器写入到处理器的对应寄存器中。

Linux PowerPC 中第一个用户进程 init 是使用系统调用 execve 将原来的进程空间进行替换的，而其他的用户进程最终都是由用户进程 init 创建的。

由此分析，我们可以发现，在 Linux PowPC 创建用户进程时，一定不能绕过系统调用 execve。因此 Linux PowerPC 在系统调用 execve 中需要对用户进程的 MSR 寄存器进行设置。系统调用 execve 的执行顺序如图 2-1 所示。

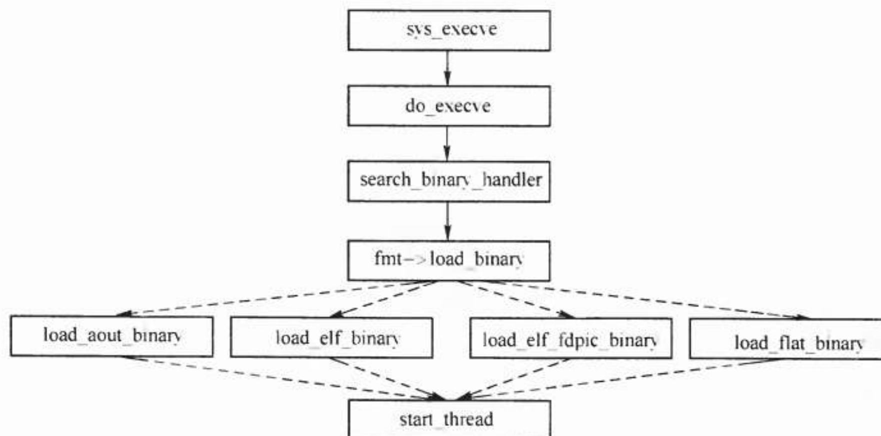


图 2-1　系统调用 execve 对 MSR 寄存器进行设置的流程

系统调用 execve 最终将调用 start _ thread 函数替换进程的正文段。图中的虚线表示 fmt ->load _ binary 函数将根据所加载的可执行文件类型，选择合适的函数加载不同格式的可执行文件，这些函数都将会调用 start _ thread 函数。

在 start _ thread 函数中将对进程描述符中存放的寄存器，包括对 MSR 寄存器进行初始化，该函数在 ./arch/powerpc/kernel/process.c 文件中。该函数将调用以下程序更改进程描述符中存放的 MSR 寄存器。

```
#ifdef CONFIG_PPC32
    regs->mq = 0;
    regs->nip = start;
    regs->msr = MSR_USER;
#else
```

用户进程获得处理器运行时，会将 PowerPC 处理器的 MSR 寄存器赋值为 regs->msr，从而保证在 Linux 系统中，用户进程可以运行在 E500 内核的用户模式中。

Linux 系统的用户进程需要进入 PowerPC 处理器的超级用户模式时，必须通过系统调用，中断或者异常才能将 MSR 寄存器的 PR 位清零，而不能直接对 MSR 寄存器进行操作，因为该寄存器只能在处理器运行在超级用户模式才能访问。Linux PowerPC 充分利用了 E500 内核提供的 MSR 寄存器将 Linux 系统的用户空间与内核空间进行了有效隔离，从而保证了在 Linux PowerPC 中用户进程不可能访问 PowerPC 处理器中的 URL。

在了解了 PowerPC 处理器的用户模式和超级用户模式的切换后，读者可以继续了解用户模式寄存器和超级用户模式寄存器。在阅读这些寄存器之前，需要提醒读者注意，在 E500 内

核中绝大多数的寄存器都是 32 位。在这些 32 位寄存器中,第 32 位为最高位,而第 63 位为最低位,而 64 位的寄存器中的第 0 位为最高位,而第 63 位为最低位。

2.2.1 URL 寄存器组

E500 内核的 URL 寄存器组与其他 PowerPC 体系的 URL 寄存器组类似。E500 内核将这些寄存器分为以下几类。

- 通用寄存器组 GPR(General-Purpose Registers)。
- 指令状态寄存器组 IAR(Instruction-Accessible Registers)。
- 性能监测寄存器组 PMR(Performance Monitor Regtsters)。在用户模式下只读。
- L1 Cache 寄存器组(L1 Cache Registers)。在用户模式下只读。
- 用户通用 SPR 寄存器组(User General Special-Purpose Registers)。
- 通用 SPR 寄存器组(General Special-Purpose Registers)。在用户模式下只读。
- Time Base Registers。在用户模式下只读。
- 多功能寄存器(Miscellaneous Registers),包括 BBEAR 和 BBTAR,用作动态分支预测。

在本节中主要介绍通用寄存器组 GPR 和指令状态寄存器组 IAR,URL 寄存器组中的其他寄存器在用户模式下只读,在超级用户模式下可以进行读写操作。

1. 通用寄存器组 GPR

E500 内核一共定义了 32 个 64 位的 GPR,GPR0～GPR31。其中 GPR 的高 32 位只能由 E500 内核的 SPE 使用,而 E500 内核的其他指令只能使用 GPR 的低 32 位,即第 32～63 位。本书不对 GPR 的高 32 位的使用方法进行说明,对此部分有兴趣的读者可以阅读 Book E 的参考手册。

E500 内核的大多数指令都可以使用通用寄存器作为操作数。E500 内核使用指令集为 32 位,在这些指令中,通用寄存器将占用 5 位,用来表示 GPR0～31。

在现代的 32 位 RISC 处理器中,通用寄存器的个数一般为 32 个,一个处理器使用的通用寄存器个数过多将占用指令的空间,过少则不能充分利用指令的空间。对于 PowerPC 处理器,32 个通用 GPR 是一个合理的数字。

如果一个程序不考虑兼容性而且所有的代码都使用汇编语言开发,编程人员就可以自定义规则将这些寄存器作为真正的通用寄存器任意使用。但是,对于一个实际的系统,程序的兼容性不可以忽略,而且绝大多数应用不会完全使用汇编语言。为此汇编程序员们必须要阅读 E500 内核的 ABI(Application Binary Interface)手册,了解如何按照规范使用 GPR 寄存器。本书将在 2.4 节中简要介绍 ABI 手册。除了这个 ABI 手册之外,Linux 系统还规定通用寄存器 GPR2 保存当前进程描述符的地址。

2. CR 寄存器

E500 内核中的 IAR 寄存器组是用来保存指令执行结果或者控制指令执行的一组寄存器,在 IAR 寄存器组中包括 CR(Conditional Register)寄存器,CTR(Counter Register)寄存器,LR(Link Register)寄存器,XER(Interger Exception Register)寄存器和 ACC(Accumulator)寄存器。

CR 寄存器用来存放指令执行后的状态,该寄存器分为 8 个字段,分别为 CR0～7,其中每个字段 由 4 位组成。这些字段被用来表示指令操作的结果。

其中整型运算指令，如整型数的加减及逻辑运算，使用 CR0 保存结果状态。

- CR0[0]，用于表示 LT(小于，当整型指令运算结果为负时置 1)。
- CR0[1]，用于表示 GT(大于，当整型指令运算结果为正时置 1)。
- CR0[2]，用于表示 EQ(等于，当整型指令运算结果为 0 时置 1)。
- CR0[3]，用于表示 SO(溢出，当整型指令运算结果溢出时置 1)。

浮点运算指令使用 CR1 保存结果状态。

- CR1[0]，用于表示 LT(小于，当浮点指令运算结果为负时置 1)。
- CR1[1]，用于表示 GT(大于，当浮点指令运算结果为正时置 1)。
- CR1[2]，用于表示 EQ(等于，当浮点指令运算结果为 0 时置 1)。
- CR1[3]，用于表示 SO(溢出，当浮点指令运算结果溢出时置 1)。

E500 内核的比较指令可以使用 CR 寄存器的全部 CRn 字段(n 从 0 到 7)保存比较指令的结束状态。在比较指令如 cmp 指令后，一般会紧跟着一条跳转指令，如 bc 指令。其中 cmp 指令可以选择使用 CR 寄存器的指定的 CRn 字段存放比较指令的结果状态，bc 指令也可以选择使用 CR 寄存器的指定的 CRn 中的状态进行跳转，一般来说 bc 指令选择的 CRn 和 cmp 中指定的 CRn 相同。比较指令使用的 CRn 字段的定义如下所示。

- CRn[0]，用于表示 LT(小于，当比较结果为负时置 1)。
- CRn[1]，用于表示 GT(大于，当比较结果为正时置 1)。
- CRn[2]，用于表示 EQ(等于，当比较结果为 0 时置 1)。
- CRn[3]，用于表示 SO(溢出，当比较结果溢出时置 1)。

比较指令和跳转指令通常由编译器从 CR 寄存器中选择合适的 CRn 字段，程序员也可以书写汇编语言，在 CR 寄存器中选择合适的 CRn。如果在比较指令和跳转指令不选择 CR 寄存器的 CRn 字段时，这些指令将使用 CR0 字段。

CR 寄存器的这一特性对于布尔运算指令，如 f=((a＞0)&& (b=1)&& (c＜0)&& (d－e))，有一定的优化作用。比如编译器可以将 a＞0 的运算结果放入 CR0，b=1 的运算结果放入 CR1，c＜0 的运算结果放入 CR2，将 d-e 的结果放入 CR3，之后使用 crand 指令合并存放这些 CRn 字段的比较结果并放入布尔变量 f 中。

除此之外，PowerPC 处理器提供的多个 CRn 字段还可以有效地避免多条比较指令在使用 CR 寄存器时产生的相关性。

我使用 gcc 编译器作了各种各样的多元判断语句的实验，发现 gcc 编译器似乎并没有使用 PowerPC 处理器中提供的 cror 和 crand 指令为多元判断语句和布尔运算做出优化。也许这是因为 gcc 编译器需要照顾其他体系结构处理器，因此只使用一些绝大多数处理器都具有的汇编指令。也许 Diab C 编译器能够充分利用 PowerPC 的指令集。对此有兴趣而且具有 Diab C 编译器的读者，可以自己做几个关于这方面的试验。

3. 其他指令状态寄存器

(1) CTR 寄存器。CTR 寄存器用来保存循环变量，并可以根据条件转移指令 bclr 的 BO 操作数自动进行减一操作。因此循环语句中需要使用 CTR 寄存器。此外 CTR 寄存器还可以用作保存 bcctr 指令的目标地址，用来实现长跳转。

(2) LR 寄存器。LR 寄存器用来存放函数的返回地址。某些转移指令可以自动将 LR 寄存器 赋值为转移之后的指令地址。每一个转移指令的编码中都有一个 LK 位。如果 LK 为

1,转移指令就会将当前指令地址加 4 对 LR 寄存器进行设置。

LR 寄存器还可以用作保存 bclr 指令的目标地址,用来实现长跳转。在应用中,bclr 指令比 bcctr 指令更常见。LR 寄存器经常被用来实现函数的调用与返回。如图 2-2 所示,假定"bl fun1"指令的地址为 0x48000000,当执行"bl fun1"指令后,LR 寄存器的值将被设置为 0x48000004。在 fun1 函数执行完毕后,需要使用 blr 指令将程序跳转到 LR 寄存器所指定的地址中。

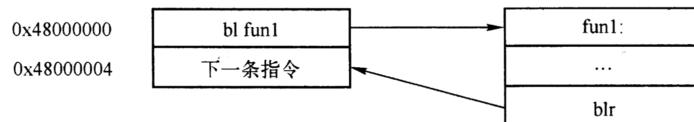


图 2-2　使用 LR 寄存器实现函数调用

(3) XER 寄存器。XER 寄存器存放整数运算操作的进位、溢出信息以及特殊的加载和存储指令,lswx 和 stswx 指令中传输的字节数。XER 寄存器共有 3 个有效位和一个有效字段,分别为 SO(Summary Overflow,第 32 位),OV(Overflow,第 33 位),CA(Carry,第 34 位)和 No. of Bytes 字段(第 57～63 位)。

其中,SO 位为 1 表示某条算术运算指令曾经将 OV 位置 1,CR 寄存器中的 SO 位就是复制了 XER 寄存器中的 SO 位,SO 位一旦被设置为有效,将不会被清除,直到程序使用 mtspr 或者 mcrxr 指令进行清除。

OV 位为 1 表示有符号数的算术运算出现溢出。PowerPC 处理器使用补码表示有符号数,并且使用双符号位(对应 PowerPC 处理器 Carry out 的 32 和 33 位)表示算术运算的结果是正数还是负数。如果双符号位为 00 表示算术运算的结果是正数;为 11 表示算术运算的结果是负数; 01 表示算术运算的结果出现正溢出;10 表示算术运算的结果出现负溢出。当计算结果出现正溢出或者负溢出时,OV 位被置 1;如果计算结果没有出现溢出,OV 位被置 0。

CA 位为 1(即 Carry out 的 32 位为 1)表示无符号算术运算产生了进位。

进位与溢出是完全不同的两个概念,大学的计算机原理的课程已经将这两个概念阐述得十分清楚。但是也许有一些程序员似乎已经忘记了这两个概念之间的区别。为此本书再次对这两个概念进行强调,进位标志表示无符号数的运算结果超出了范围,但是运算结果依然正确;而溢出标志表示有符号数的运算结果超出了范围,运算结果已经不正确了。在算术运算中,应该使用进位还是溢出由程序员决定。具体地说,如果参加算术运算的操作数是无符号数,程序员应该关心进位位;如果参加算术运算的操作数是有符号数,程序员应该关心溢出位。

No. of Bytes 字段用来存放 lswx 和 stswx 指令中传送的字节数。

(4) ACC 寄存器。该寄存器是一个 64 位寄存器,用来保存 E500 内核 SPE 指令的乘加指令 MAC(Multiply Accumulate)的计算结果。

2.2.2　SLR 寄存器组

SLR 是 E500 内核中重要的寄存器组。该组寄存器中包括 CPU 配置、内存管理、中断控制、SPR 寄存器等一系列寄存器。在 E500 内核中,主要使用两条指令用于操作这些寄存器,分别是 mtspr 和 mfspr 指令,mtspr 的作用是对 SLR 中的寄存器进行赋值,mfspr 的作用是对

SLR 中的寄存器进行读取。E500 内核对 SLR 寄存器进行编址访问，如 TBL 寄存器的地址为 268，而 TBU 寄存器的地址为 269，在 PowerPC 使用 spr268 和 spr269 表示 TBL 和 TBU 寄存器。E500 内核使用“mtspr 268，r3”指令对 TBL 寄存器进行赋值，“mfspr r3，268”指令读取 TBL 寄存器。

1. MSR 寄存器

MSR 寄存器用来设置 E500 内核的当前使用状态。这个寄存器基本上是初始化 E500 内核时第一个被配置的寄存器。该寄存器在 E500 内核中非常重要。MSR 寄存器属于 SLR 寄存器组，但是 mtspr 和 mfspr 指令不能访问该寄存器，E500 内核使用专门的指令 mtmsr 和 mfmsr 指令对 MSR 寄存器进行访问。上文对 MSR 的 PR 位进行了详细介绍，下面将介绍其他位。

- UCLE(User-mode Cache Lock Enable，第 37 位)。该位为 1 时，程序运行在 PowerPC 处理器用户模式时，也可以锁定 Cache 行。
- SPE(SPE Enable，第 38 位)。该位为 1 时，使能 E500 内核的 SPE 部件。
- WE(Wait State Enable，第 45 位)。该位与 HID0 的寄存器的 DOZE，NAP 和 SLEEP 位联合使用可以设置 E500 内核进入等待模式。
- CE(Critical Enable，第 46 位)，ME(Machine Check Enable，第 51 位)，DE(Debug Interrupt Enable，第 54 位)位用来使能或者关闭 Critical 异常、Machine check 异常和调试中断。
- EE(External Enable，第 48 位)。此位为 1 时，将使能外部中断，为 0 时屏蔽外部中断。当处理器进入外部中断处理时会自动关闭此位，屏蔽外部中断，以防止中断嵌套。支持中断重入的驱动程序需要在程序中重新使能此位。许多操作系统需要频繁的改动 EE 位。为此，E500 内核专门设置了 wrtee 和 wrteei 指令对 MSR 寄存器的 EE 位进行修改。
- IS，DS(第 58，59 位)。E500 内核支持两个地址空间，分别为地址空间 0 和地址空间 1。IS 位为 1 表示当前程序使用的指令空间 1；IS 位为 0 表示当前程序使用的指令空间 0；DS 位为 1 表示当前程序使用的数据空间 1；DS 位为 0 表示当前程序使用的数据空间 0。

E500 内核的内存管理与其他 PowerPC 内核有较大的差别，它不能关闭虚实地址转换功能。而许多 PowerPC 内核，如 603E 内核中可以关闭虚实地址转换功能，在 603E 中，IR，DR 位与 E500 内核的 IS 和 DS 两位相对应。但是在 603E 内核中 IR，DR 位用来使能和禁止地址与数据空间的虚实地址转换。

MSR 寄存器还有一些其他位，本书对此不再一一叙述，这些位的详细描述见 E500 内核参考手册。

2. 其他 SPR 寄存器

(1) PVR 寄存器(Processor Version Register)。PVR 寄存器用来存放当前处理器使用的 PowerPC 内核版本号。PID 寄存器与处理器内核版本对应如表 2-1 所示。

表 2-1 PowerPC 的版本号

Soc 版本号	E500 版本号	PVR 寄存器
1.0	1.0	0x80200010
1.1	2.0	0x80200020
2.0	2.0	0x80210010

(2) SVR 寄存器(System Version Register)。该寄存器用来存放当前处理器的

版本号。如 MPC8555E 的 SVR 寄存器为 0x80790010,MPC8541E 的 SVR 寄存器为 0x807A0010,MPC8560 的 SVR 寄存器为 0x80700010。

PQIII 处理器的 SVR 寄存器的低 16 位用来表示 PQIII 芯片的版本号。如 MPC8560 V1.0 版本的芯片低 16 位为 0x0010,而 V2.0 版本的芯片低 16 位为 0x0020。PowerPC 工程师在进行程序设计时,必须关注 SVR 和 PVR 寄存器的值,以确定正在使用的 SoC 的版本。

Freescale 为不同版本的 PowerPC 处理器提供了一个 Errata(勘误表)。这个 Errata 与数据手册同样重要,在设计 PowerPC 处理器系统时,一定要仔细阅读这些 Errata。

到目前为止,我几乎没有发现 Freescale 任何一个版本的 PowerPC 处理器没有 Errata。而且对于有些 Errata,Freescale 恐怕已经没有修正的计划了。

因此,使用 PowerPC 处理器的用户必须要留意你们可能使用的 PowerPC 处理器不一定是最新版本的。工程师们一定要选择合适的 Errata,而不一定是最新的 Errata。

(3) HID0 寄存器(Hardware Implementation-Dependent Register 0)。HID0 寄存器的 EMCP 位为 1 时使能将使能 MCP # 引脚。

DOZE,NAP,SLEEP 位用于使能与功耗管理有关的引脚。

TBEN 位用于使能 TB(Time Base)和 DEC(Decrementer)寄存器,TB 和 DEC 寄存器可以用来纪录处理器的运行时间。其中 TB 寄存器还可以作为 Benchmark 程序等其他与系统性能有关的测试程序使用的计数器。而 DEC 寄存器经常被用作操作系统的系统时钟中断计数器。

SEL_TBCLK 位用于选择是使用处理器的系统时钟还是外部 RTC 的时钟源,由于处理内部的系统时钟的误差较大,因此需要精确时钟的系统最好使用外部 RTC 时钟。一般来说一个处理器系统中都有独立的 RTC 器件。

(4) HID1 寄存器(Hardware Implementation-Dependent Register 1)。HID1 寄存器的 PLL_CFG 字段用于记录 E500 系统时钟(CCB Clock)与 E500 内核时钟间的比率。用户还可以访问 PQIII 处理器的另外一个内存映像的寄存器 PORPLLSR 获得这个比率。

程序员在编程时多使用 PORPLLSR 而不是使用 HID1 中的字段,HID1 寄存器中的 PLL_CFG 字段的定义可能随处理器的不同而有所变化。HID1 寄存器的其他位还可设置系统总线的奇偶校验和地址广播等其他功能。

(5) 通用 SPR 寄存器组。这组寄存器包括 SPRG0-SPRG7 与 USPRG0,此组寄存器除 USPRG0 外,其他寄存器在用户模式下只读。

SPRG0~7 是 E500 内核中的特殊功能寄存器。PowerPC 处理器处于超级用户模式下可以对 SPRG0~2 寄存器进行读写操作,处于用户模式时不能操作此组寄存器;PowerPC 处理器处于超级用户模式下可以对 SPRG3~7 寄存器进行读写操作,处于用户模式下可以对此组寄存器进行读操作。

需要注意的是 SPRG3~7 寄存器有两个地址,一个用于处理器在用户模式或超级模式进行读操作,另一个用于处理器在超级模式状态进行读写操作。例如 SPRG4 寄存器共有两个编号 260 和 276,其中 260 编号的特殊寄存器用作超级模式或者用户模式进行读操作,编号为 276 的特殊寄存器用作超级模式下进行读写操作。在许多操作系统中使用 SPRG4W 和 SPRG4R 分别表示这两个寄存器。

这组寄存器主要用于操作系统的某些特定操作。如在 Linux PowrPC 中,SPRG3 用来保存当前进程的 thread 参数的地址,其他的 SPR 寄存器也可以供操作系统开发人员选用,如这

些寄存器可以在 Linux PowerPC 的中断处理程序中保存即将要使用的 GPR 寄存器。系统程序员需要慎重地使用这组寄存器，一般来说应用软件工程师没有使用这组寄存器的必要。SLR 寄存器组中有关内存管理和中断系统将陆续在下文中介绍。

2.3 E500 内核的常用指令集

E500 内核的指令长度都是 32 位。E500 内核的指令采用大端编码方式，指令的第 0 位是 MSB(Most Significant Bit)，第 31 位是 LSB(Least Significant Bit)。

在 E500 内核中，指令的高 6 位字段(第 0～5 位)被称为 OPCD 字段(Primary Opcode Field)。根据 OPCD 字段的不同，PowerPC 的指令集可以分为以下几大类。

2.3.1 I-Form 类指令

在 E500 内核中，无条件转移指令使用这种指令格式，这种指令格式相对较为简单。I-Form 类指令格式如下所示：

0～5	6～29	30	31
OPCD	LI	AA	LK

I-Form 类指令共包含以下几种指令：

- b　LI//AA=0，LK=0
- ba　LI//AA=1，LK=0
- bl　LI//AA=0，LK=1
- bla　LI//AA=1，LK=1

这类指令格式支持 AA 位和 LK 位 。

AA 位用来表示当前跳转指令使用绝对地址还是相对地址进行跳转。该位为 0 表示 LI 中存放的是相对地址(该相对地址是有符号整数，用户分析机器码时需要注意)，即将要跳转的指令地址是当前指令地址与 LI＊4 的和；为 1 表示 LI 中存放的是绝对地址，即将要跳转的指令是 LI＊4。AA 位为 1 的无条件转移指令很少被用户使用。

LK 位用来表示指令执行后是否修改 LR 寄存器。LK 为 1 时，表示转移指令执行后将当前指令的下一条指令的地址存放在 LR 寄存器中，如果 LK 为 0 时，LR 寄存器将不会被修改。

在汇编语言中 bl 指令使用最多，bl 指令可用作函数的调用指令，还可以用来获得下一条指令的有效地址。例如在 Linux PowerPC 的内核入口地址使用 bl 指令获得当前程序运行的地址，该程序如下所示。

```
        bl invstr                   /* Find our address */
invstr: mflr r6                     /* Make it accessible */
```

这段程序使用了 bl 指令获得 invstr 的地址，并将其存放到通用寄存器 r6 中。

2.3.2 B-Form 类指令

E500 内核的条件转移指令使用这一指令格式，这一指令格式也支持 AA 位和 LK 位 。B-Form 类指令格式如下所示：

0~5	6~10	11~15	16~29	30	31
OPCD	BO	BI	BD	AA	LK

B-Form 类指令共包含以下几种指令：

- bc　　BO，BI，BD　//AA=0 LK=0
- bca BO，BI，BD　//AA=1 LK=0
- bcl　　BO，BI，BD　//AA=0 LK=1
- bcla　　BO，BI，BD　//AA=1 LK=1

BI 字段用来确定使用 CR 寄存器 CRn 的字段中哪个状态位作为指令跳转的条件。其中 BI 的第 0~2 字段用来选择使用哪个 CRn 作为状态字段，该字段为 0 时表示使用 CR0 字段作为转移指令使用状态字段，为 1 时表示使用 CR1 字段作为转移指令使用的状态字段。而 BI 的第 3~4 字段选择使用什么条件作为转移条件，如下所示：

BI[3:4]	描述
00	使用 LT 状态位用做指令转移条件
01	使用 GT 状态位用做指令转移条件
10	使用 EQ 状态位用做指令转移条件
11	使用 SO 状态位用做指令转移条件

BO 字段用来表示指令根据什么条件进行跳转。

(1) 第 0 位。如果此位为 1，bc 类指令将不根据 CR 寄存器的状态进行条件转移，此时 bc 指令只能根据 CTR 寄存器的值为 0 或者不为 0 决定是否进行条件转移。程序使用简单的 Loop 循环语句时，可以将此位置 1。此位为 0 时，表示 bc 类指令将根据 CR 寄存器的相应字段和存放在 BI 中的条件决定是否进行跳转。

(2) 第 1 位。如果此位为 1，则当指令执行的条件为真时进行转移，否则当指令执行的条件为假时进行转移，此位可以用来实现布尔表达式中的非操作。在 CR 寄存器的 CRn 字段中只定义了 LT，GT，EQ 位，这些位可以实现 "小于"、"大于"或"等于"这些布尔条件，如果程序需要实现"不小于"、"不大于"和"不等于"这些布尔条件时，需要使用此位。例如将此位置为 0，并将 BI[3:4]置为 00，即可实现"不小于"这种转移条件。

(3) 第 2 位。如果该位为 1，执行 bc 指令时，CTR 寄存器的值保持不变；如果该位为 0，执行 bc 指令时，CTR 寄存器将做自减操作。

(4) 第 3 位。如果此位为 1，表示当 CTR 寄存器为 0 时进行条件转移。当此位为 0 时，表示当 CTR 寄存器为 0 时进行条件转移。

(5) 第 4 位："y"位。此位用来支持静态分支预测功能。此位为 1 时表示此转移语句预先将被判断为执行转移功能，处理器将预取转移指令目标地址后的指令，并将这些指令放入缓冲队列；当此位为 0 时表示此转移指令被预先判断为不被执行，因此不需要做额外的指令预取操作。

E500 内核不支持指令的静态预取操作，因此不支持这一位。E500 内核使用指令动态预取来增强转移语句命中的效率。

E500 内核的 B-Form 类指令集一共使用了 4 条指令描述了所有的条件转移指令。如使用"bc 16，0，BD"表示将 CTR 寄存器进行自减操作，如果 CTR 不为 0 则进行跳转；使用"bc 4，

0, BD"表示运算或比较结果为 0 时进行跳转。这种指令实现方式便于 IC 设计工程师对指令集进行设计。但是对于使用汇编语言的软件工程师,编写这种格式的指令是十分困难的。

为此 E500 内核使用了许多助记符用来表示各种条件跳转指令,如使用"bdnz"助记符表示"bc 16, 0, BD",使用'beq'助记符表示"bc 4, 0, BD"。常用的助记符还有 lt(小于),le(小于等于),eq(等于),so(溢出)等。

在使用条件转移指令的助记符进行编程时,可以指定使用 CR 寄存器的哪个字段进行转移,如"beq crS, BD"。使用助记符也可以控制程序是否进行静态分支预测,为此助记符后面可以加上"+"和"-"两个符号。'+'表示转移被静态预测为真,选择转移;"-"表示转移被静态预测为假,如"beq+"或"beq-"。

需要再次提醒读者注意的是,在 PowerPC 处理器中,转移指令非常少,常用的条件转移指令只有 bc 和 bcl。其他的条件转移指令,如 beq,bdnz,ble 指令等,不过是助记符而已,并不是真正的转移指令。

2.3.3 SC-Form,D-Form 与 DS-Form 类指令

SC-Form 类指令用来实现系统调用指令。在 Linux 系统中,系统调用是用户程序进入内核空间的一种方式,E500 内核使用"sc"指令实现系统调用。这类指令较为简单,此处不再对此进行说明。

D-Form 类指令主要包括以下几类指令。

- 对存储器(包括寄存器)进行读写的指令。
- 立即数的算术运算和逻辑运算指令。

由上所示,D-Form 类指令包括对存储器的直接读写操作指令,也包括经过加减,逻辑运算后再对存储器进行操作的指令。

D-Form 类指令由一个 16 位的立即数,两个寄存器索引(10 位)和 6 位的 opcode 组成。其中 16 位的立即数可以作为地址偏移也可以作为算术运算的操作数。D-Form 类指令格式如下所示:

0~5	6~10	11~15	16~31
OPCD	RS	RA	D

RS 字段用来索引存放该指令运算结果的寄存器,RA 字段用来索引存放该指令所需要数据源的寄存器,而 D 用来存放该指令所需要的另外一个数据源,立即数。在 D-Form 类指令中,一定包含一个立即数。

D-Form 类的典型指令有以下几种:

(1)"lwz RT, D(RA)"。lwz 指令将从寄存器 RA+D 指定的地址中读取一个 32 位数据,然后这个数据传递给寄存器 RT。此外还有对 16 位,8 位数据进行读取的指令 lbz,lhz 和 lha。

(2)"stw RS, D(RA)"。stw 指令将寄存器 RS 中的 32 位数据内容写入寄存器 RA+D 所指定的地址中去。此外还有对 16 位,8 位数据进行读取的指令 stb,sth 指令。

(3)"lwzu 与 stwu 指令"。这两条指令的使用方法与 lwz 和 stw 指令的格式相同,只是这类指令在执行后会将 RA 寄存器的值更新为 RA+D。这类指令对于实现数据栈的压栈和出栈操作有所帮助。

如指令“stwu r1, S _ FRAME (r1)”可以将寄存器 r1 的值放入 r1 + S _ FRAME 后的同时将寄存器 r1 的值修改为 r1 + S _ FRAME。从而程序员只使用一条指令即完成了将数据压入堆栈所需要的一些基本操作。

PowerPC 处理器没有专门的堆栈指针寄存器，而是使用寄存器 r1 模拟堆栈指针寄存器。关于 PowerPC 栈帧结构的知识请参考 2.4.3 节。此类指令中也有对 8 位和 16 位数据进行操作的指令 sthu，lhzu，lbzu 和 stbu。

(4) lmw RT, D(RA)。lmw 指令将 RA + D 地址的数据依次传递到 RT～R31 中，所传递 32 位数据的数量为 31-T 个。

(5) stmw RS, D(RA)。stmw 是将 RT～R31 的数据依次传递到 RA + D 的地址中去。使用这种批量传送时务必要注意存储器边界的检查和所使用的通用寄存器是否需要进行备份操作。

(6) “cmpi BF, L, RA, SI”与“cmpli BF, L, RA, UI”。cmpi 和 cmpli 指令将寄存器 RA 与立即数 SI/UI 进行比较，然后将比较指令产生的状态放入 BF 所指定的 CR 寄存器的不同字段里，CR 寄存器里有 8 个 CRn 子段可以由 3 位的 BF 指定。L 用来表示是进行 32 位还是 64 位比较，对于 E500 内核，L 始终为 0。cmpi 和 cmpli 的区别在于一个是算术比较（带符号位），一个是逻辑比较（不带符号位）。

(7) “tw TO, RA, RB”。此指令被称为自陷（Trap）指令，该指令对一些 Trap 条件进行测试，如果条件成立，处理器将进入系统的 Trap 程序，然后对这些 Trap 事件进行处理。

在 E500 内核中，TO 字段一共有 5 位，第 0～4 位分别表示 LTS（有符号数比较，小于）、GTS（有符号数比较、大于）、EQ（等于）、LT（无符号数比较，小于）和 GT（无符号数比较，大于）。在使用 tw 指令时，用户可以根据需要对 TO 字段进行设置，如将 TO 设置为 1，表示如果 RA 寄存器中的无符号数值大于 RB 寄存器中的数值，则处理器进入 Trap 处理程序。

当处理器执行“tw 31, r0, r0”时，处理器将无条件进行 Trap 处理程序，“trap”助记符等效于“tw 31, r0, r0”。E500 内核使用 Program Interrupt Exception 异常处理函数处理 Trap 事件。

此外，D-Form 类指令还包含了许多用于算术和逻辑运算的指令，这些指令都要求使用一个立即数，同时 E500 内核的 D-Form 指令在进行计算时还可以将立即数左移 16 位。

这样做的主要目的是为了处理 32 位立即数。E500 内核中一条指令只有 32 位，因此必须通过两条指令才能将一个 32 位的立即数赋值到一个通用寄存器中。此时程序可以先处理高 16 位，然后再将高 16 位左移 16 位，与低 16 位相或就可以完成一个 32 位立即数的赋值操作。以下程序可以将 32 位的立即数 start _ kernel 复制到通用寄存器 r6 中。

```
lis  r6, start _ kernel@h
ori  r6, r6, start _ kernel @l
```

D-Form 类指令集定义了许多有关算术和逻辑运算的指令。在这些算术指令的尾部还有许多后缀，如“.”，“c”，“o”。这些后缀又可以相互组合。

如 addi 指令就有许多衍生指令“addi.”，“addic”，“addic.”，“addio”等等。这些后缀的含义如下所示。

- 后缀“.”表示 addi 指令的结果将更新 CR 寄存器。
- 后缀“c”表示 addic 指令的结果将影响 CA 位。

● 后缀“o”表示 addo 指令的结果将更新 SO 和 OV 位。(注意在 XO-Form 类指令包含带有后缀“o”的指令,而 D-Form 类指令没有带有后缀“o”的指令)。

对于任何程序员来说,编写汇编程序都相当困难。在 PowerPC 处理器中,编写汇编程序更加费力。如果程序员必须编写汇编程序,可以首先编写一个 C 程序,然后反汇编出来,再加以修改。不过即便如此,PowerPC 程序员们最好也要了解一些用作算术运算的汇编指令。

DS-Form 类指令可以用来对双字(64 位数据)进行处理。主要的指令有 lwa(低 32 位读操作,与 lwz 指令在计算数据有效地址时有些不同),ld(64 位读操作),std(64 位写操作),ldu(64 位读更新操作),stdu(64 位写更新操作)。与 D-Form 类指令类似,DS-Form 类指令也包含一个立即数。E500 内核不支持 DS-Form 类指令。

2.3.4 X-Form 类指令

E500 内核中 X-Form 类指令的数量最多。D-Form 类指令和 DS-Form 类的指令中的每一条指令对应一条在 X-Form 类指令中的指令。除此之外,X-Form 类指令还有一些专用的指令。X-Form 类典型的指令格式如下所示:

0~5	6~10	11~15	16~20	21~30	31
OPCD	RS/RT	RA	RB	XO	Rc

可见,X-Form 类指令与 D-Form 类指令格式相似,只是 X-Form 类指令将 D-Form 类指令的 D 字段拆分为 RB、XO 字段和 Rc 位。

X-Form 类指令格式中的 RB 和 RS 字段存放源操作数寄存器的索引,RT 字段用来存放目的操作数寄存器的索引,而 RA 字段既可以存放源操作数寄存器的索引,也可以存放目的操作数寄存器的索引。XO 字段用来存放 X-Form 类指令扩展的 OPCD。Rc 字段为 1 表示当前指令的运行结果将改变 CR 寄存器,具有“.”后缀的指令其 Rc 位为 1。

X-Form 类中的许多指令与 D-Form 类中的指令一一对应,用法也基本相同,只是将 D-Form 类指令中的立即数替换成为 RB,XO 字段和 Rc 位。X-Form 类的典型指令如下所示。

(1) 存储器访问类指令。如 lbzx, lhzx, lhax, lwzx, stbx, sthx, stwx, lbzux, lhzux, lhzux, lwzux, stbux, sthux, stwux 指令等。这些指令与 D-Form 类的 lbz, stb 等一系列指令一一对应,只是多了“x”指令后缀。与 D-Form 指令不同,这一类指令在进行地址计算时,不使用立即数,而是将立即数替换成寄存器。相比较而言,D-Form 类的存储器访问指令更加常用。

(2) 字节序列交换指令。lhbrx, lwbrx, sthbrx, stwbrx 指令。这一类指令的主要作用是调整字节序列。如“lhbrx RT, RA, RB”指令是将 RA + RB 地址(或者 RB 寄存器)存放的 16 位数据的高 8 位和低 8 位进行交换然后将结果存放入 RT 寄存器中。

(3) 字符串操作指令。lswi, lswx, stswi 和 stswx 指令。这一系列指令可以对字符串进行操作。使用此类指令时需要注意,这些指令将隐式地使用一些连续的通用寄存器资源。

(4) 比较类和 Trap 指令。如 cmp, cmpl, tw 指令,这些指令的使用方法与 D-Form 类中对应指令的使用方法类似。

(5) 逻辑算术运算类指令, and, or, xor, nand, nor, eqv 等等。这些指令与 D-Form 类的同类指令对应。

(6) “cntlzw RA, RS”指令。该指令用来找出 RS 寄存器中第一个不是 0 的位,然后将此位

的位序存入 RA 中。该指令可以用来实现 LOG2N 运算，Linux PowerPC 充分利用了这条指令有效的提高了 FFB 算法的检索速度。

(7) “popcntb RA，RS”指令。该指令计算 RS 寄存器中存放的 8 个字节中每一个字节里有多少个 1，并将相应的结果存放在 RA 的对应字节中。E500 内核不支持该指令。

(8) 移位类指令。如 sld，slw 指令等。注意 D-Form 类没有移位类指令。

2.3.5 XL-Form 类指令

XL-Form 类指令支持条件转移指令，与 B-Form 类条件转移不同，此类指令使用 LR 寄存器或者 CTR 寄存器，而不是使用 16 位立即数进行跳转，因此可以用来实现 32 位长跳转。XL-Form 类条件转移指令的格式如下所示：

0~5	6~10	11~15	16~18	19~20	21~30	31
OPCD	BO	BI	~	BH	16 或者 528	LK

所支持的指令有以下几种：

```
bclr   BO,  BI,  BH        //LK=0,第 21~30 字段为 16
bclrl BO,  BI,  BH         //LK=1,第 21~30 字段为 16
bcctr BO,  BI,  BH         //LK=0,第 21~30 字段为 528
bcctrl     BO,  BI,  BH         //LK=1,第 21~30 字段为 528
```

X-Form 类指令的 BO 和 BI 字段与 B-Form 类指令中的 BO 和 BI 的定义相同，LK 位为 1 时表示跳转指令执行后 LR 寄存器指向下一条指令的地址(当前指令地址加 4)，BH 字段用于静态分支预测，E500 内核的 XL-Form 类指令不支持这一字段。

当条件满足时，bclr 和 bclrl 指令使用 LR 寄存器进行长跳转，而 bcctr 和 bcctrl 指令使用 CTR 寄存器进行长跳转。

与 B-Form 类指令类似，XL-Form 类条件转移指令也使用了许多助记符，如用“blr”表示“bclr 20，0”。XL-Form 类指令可以和 I-Form 类指令协作完成 C 函数的调用和返回。XL-Form 指令还可以支持 CR 寄存器不同的字段的与、或、异或、同或等操作，也可以对 CR 寄存器相应进行读取操作。

XL-Form 类操作 CR 寄存器指令的格式如下所示：

0~5	6~10	11~15	16~20	21~30	31
OPCD	BT	BA	BB	257/449/193/225/0	~

所支持的指令有以下几种：

```
crand BT,  BA,  BB  //第 21~30 字段为 257,BT ← BA & BB
cror   BT,  BA,  BB  //第 21~30 字段为 449,BT ← BA | BB
crxor BT,  BA,  BB  //第 21~30 字段为 193,BT ← BA⊕BB
crnand      BT,  BA,  BB  //第 21~30 字段为 225,BT ← ! (BA & RB)
mcrf   BF,  BFA  //第 21~30 字段为 0,CRbf← CRbfa
```

2.3.6 XFX-Form,XFL-Form,XS-Form,XO-Form 与 A-Form 类指令

XFX-Form 类包括 mtspr,mfspr,mtcfr,mfcr,mtocrf 指令。用于访问 SPR 寄存器和 CR 寄存器。XFX-Form 类条件转移指令的格式如下所示:

0~5	6~10	11~20	21~30	31
OPCD	RT	spr/tbr	XO	~

XO-Form 类指令用来支持带进位的算术运算指令和乘除法指令。XO-Form 类条件转移指令的格式如下所示:

0~5	6~10	11~15	16~20	21	22~30	31
OPCD	RT	RA	RB	OE	XO	Rc

XO-Form 类的典型指令有以下几类:

- addo.,subfo.,addco.,subfco. 指令,此类指令的结果将影响 CA,SO,OV 位和 CR0 字段。
- addeo.,subfeo.,addzeo.,subfzeo. 指令,此类指令除了可以影响 CA,SO,OV 位和 CR0 字段外,还可以将 CA 位参与加减运算。
- mullw,divw 指令,此类指令用作乘除运算。

A-Form 类指令用作浮点运算,典型指令有 fadd,fsub,fmul 和 fdiv 指令。E500 内核不支持此类指令,同时 E500 内核也不支持 XS-Form 类指令。

2.3.7 M-Form 类指令

M-Form 类指令的主要作用是对选定的字段进行循环左移并做一些相应的掩码操作。该类指令是 PowerPC 指令集的精华,包含了一组非常强大的指令集。汇编语言工程师必须要熟练掌握该类指令。

在这类指令中,rlwimi 指令最为常用,该指令格式如下所示:

0~5	6~10	11~15	16~20	21~25	26~30	31
OPCD	RS	RA	SH	MB	ME	Rc

该指令的使用格式为"rlwimi RA,RS,SH,MB,ME",其功能描述如下所示。

```
n ← SH
r ← ROTL32((RS)32:63, n)
m ← MASK(MB+32, ME+32)
RA ← r & m | (RA)&¬m
```

该段描述的详解如下:

- rlwimi 指令首先将存放在 RS 寄存器中的数据循环左移 SH 位,并将此结果赋于 r。
- 之后使用 MASK 函数计算掩码 m,MASK(MB+32, ME+32)表示将一个数据的 32~63 位中的第 MB+32~ME+32 字段置 1,其他字段置 0,然后再将此数值赋于 m。
- 如果 ME 的数值小于 MB,则将第 ME+32 与 MB+32 之间的字段置 0(不包括 ME+32 和 MB+32 位),其他字段置 1。最后将 r&m | (RA)&¬m 的结果赋于 RA。该指令的

图解如图 2-3 所示。

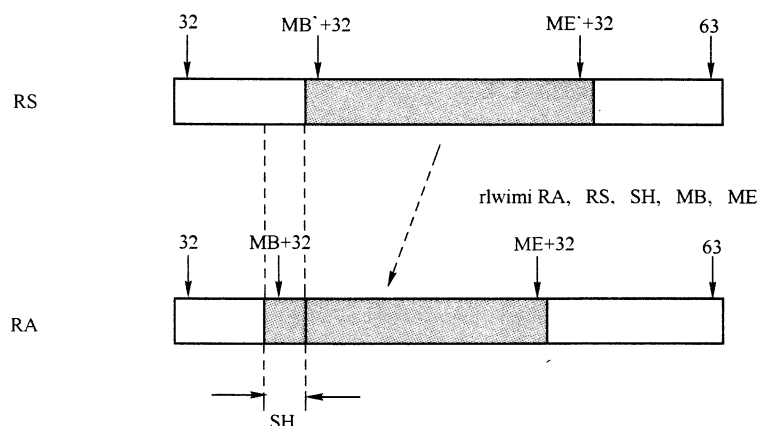


图 2-3　ME 大于或者等于 MB 时的 rlwimi 指令

当 ME 大于或者等于 MB 时 rlwimi 指令的执行流程，即将 RS 寄存器中的阴影部分平移到 RA 寄存器中，RA 寄存器不在阴影部分的字段保持不变。

当 ME 小于 MB 时 rlwimi 指令如图 2-4 所示。

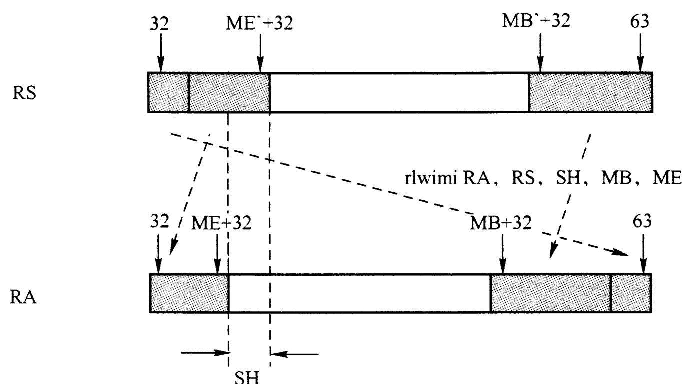


图 2-4　ME 小于 MB 时的 rlwimi 指令

M-Form 类指令还包含另一条重要的指令 rlwinm。其指令格式、使用方法与 rlwimi 指令完全相同。

这两条指令的主要区别在于指令运行结束后对 RA 寄存器的赋值不同。

rlwinm 指令最后将 RA 寄存器赋值为 r & m 而不是 r & m | (RA)&¬m，即 rlwinm 指令结束后图 2-3 和 2-4 中的空白字段将被填充为 0。该指令的主要作用是提取 RS 寄存器中的某些字段，然后将这一字段放到 RA 寄存器的某个位置中。这两条指令在 Linux PowerPC 中经常使用，希望读者认真理解这两条指令的含义。

M-Form 类指令还包含一条指令 rlwnm，该指令与 rlwinm 指令的区别在于该指令将左移位数保存在寄存器 RB 中而不是立即数 SH 中。

PowerPC 体系结构中还定义了对双字进行操作的 MD-Form 和 MDS-Form 类指令，但是这类指令没有在 E500 内核中实现。

PowerPC 处理器采用 RISC 结构，因此它的基本指令集并不很多，只是这些基本指令可以衍生出十分庞大的指令集，而令初学者望而生畏。

本节将 PowerPC 的指令集进行了简单的归纳，希望对读者有所帮助。同时希望读者能够深入地学习 PowerPC 处理器的指令集。

对于软件程序员，了解一个处理器，首先需要学习这个处理器的指令集。PowrePC 处理器的指令集是软件程序员需要掌握的基础知识，程序员至少需要浏览一遍 PowerPC 处理器的所有指令集。学习 PowerPC 的指令集并没有什么捷径，有时程序员会有很枯燥的感觉。

2.4 E500 内核的 ABI

E500 内核的 ABI 中定义了 E500 内核中支持的数据类型；通用寄存器的使用规则；PowerPC 栈帧结构；ELF 文件的组成等一系列与编译器相关的内容。

PowerPC 的编译器必须遵守 ABI 手册，并按照 ABI 的规定来产生目标代码。进行 C 语言与汇编语言混合编程的程序员必须掌握 ABI。

2.4.1 E500 内核使用的数据类型

ABI 规定 E500 内核的数据定义如表 2-2 所示。

表 2-2　E500 内核的数据类型定义

<table>
<tr><th>类　型</th><th>ANSI C</th><th>大　小</th><th>对　界</th></tr>
<tr><td rowspan="16">整型数据</td><td>char</td><td rowspan="2">1B</td><td rowspan="3">8 位</td></tr>
<tr><td>unsigned char</td></tr>
<tr><td>signed char</td><td>1B</td></tr>
<tr><td>short</td><td rowspan="2">2B</td><td rowspan="3">16 位</td></tr>
<tr><td>signed short</td></tr>
<tr><td>unsigned short</td><td>2B</td></tr>
<tr><td>int</td><td rowspan="5">4B</td><td rowspan="7">32 位</td></tr>
<tr><td>signed int</td></tr>
<tr><td>long int</td></tr>
<tr><td>signed long</td></tr>
<tr><td>enum</td></tr>
<tr><td>unsigned int</td><td rowspan="2">4B</td></tr>
<tr><td>unsigned long</td></tr>
<tr><td>long long</td><td rowspan="3">8B</td><td rowspan="3">32 位</td></tr>
<tr><td>signed long long</td></tr>
<tr><td>unsigned long long</td></tr>
<tr><td rowspan="2">指针类型</td><td>any *</td><td rowspan="2">4B</td><td rowspan="2">32 位</td></tr>
<tr><td>any(*)()</td></tr>
</table>

（续）

类　型	ANSI C	大　小	对　界
浮点类型	float	4B	32 位
	double	8B	64 位
	long double	16B	128 位

E500 内核不支持浮点运算，程序员可以使用定点运算模拟浮点运算，或者使用 E500 内核中的 SPE 部件实现浮点运算。双精度(double)和多精度(long double)浮点运算的速度较慢，占用的资源也较多。因此一般来说，很少有处理器直接支持双精度和多精度浮点运算。

对于不支持双精度和多精度浮点运算的处理器，用户需要编写程序对双精度和多精度浮点进行处理，并采取一些特殊的方法提高双精度和多精度浮点运算的效率。此外，许多用户不直接使用 double 和 long double 类型描述双精度和多精度浮点数，而是采用某些自定义的数据结构。

2.4.2 E500 内核寄存器的使用

E500 ABI 手册规定如何使用 E500 内核的 URL 寄存器。用户在进行编程时，需要遵循 ABI 中的这些规定。

- GPR0。E500 ABI 规定普通用户不能使用此寄存器。GCC 编译器使用 GPR0 寄存器保存 LR 寄存器。Linux PowerPC 还使用该寄存器传递系统调用号码。该寄存器是一个易失型寄存器(volatile)。
- GPR1。E500 ABI 规定该寄存器保存堆栈的栈顶指针，PowerPC 处理器没有设置独立的栈顶指针寄存器(Stack Pointer，SP)。
- GPR2。E500 ABI 规定一般用户不能使用此寄存器。Linux PowerPC 使用 GPR2 寄存器用来保存当前进程的进程描述符地址。
- GPR3～GPR4。E500 ABI 使用这两个寄存器保存程序的返回值。基于 E500 内核的函数返回值最多为 64 位。
- GPR3～GPR10。E500 ABI 首先使用 GPR3～GPR10 共 8 个寄存器传递函数的参数，当函数的参数多于 8 个时，E500 ABI 使用堆栈进行参数传递。为提高函数的调用效率，程序员需要尽量将函数所需参数的个数控制在 8 个之内。
- GPR11～GPR12。E500 ABI 规定一般用户不能使用此寄存器。Linux PowerPC 有时使用这两个寄存器存放临时变量。GCC 编译器没有使用这两个寄存器。
- GPR13。E500 ABI 规定该寄存器保存 sdata 段的基地址指针。Linux PowerPC 可以在系统初始化阶段时使用该寄存器存放临时变量。编译器有时会根据某些规则将一些常用的数据放入 sdata 或者 sbss 段中。应用程序对 sdata 或者 sbss 段的数据访问与对数据段 data 和 bss 段的访问机制不同，访问 sdata 段内的数据速度更快些。
- GPR14～GPR31。E500 ABI 使用这些寄存器存放一些临时变量，在应用程序中可以自由使用这些寄存器资源。有些人喜欢倒序使用这些寄存器，即首先使用 GPR31。有些编译器使用寄存器 GPR31 指向系统环境变量。

2.4.3 E500 内核的栈帧结构

PowerPC 处理器没有在指令级别上支持堆栈，比如没有专门的堆栈类寄存器及访问堆栈的指令。许多处理器都没有专门的指令支持堆栈结构。但是大多数的程序员仍然习惯性地寻找类似 Intel 奔腾体系结构的 push，pop 指令和独立的堆栈指针 SP。PowerPC 处理器使用存储器访问指令，如 stwu 和 lwzu 指令替代 push 和 pop 指令，E500 ABI 规定使用寄存器 GPR1 模拟 SP 寄存器。

与 Pentium 处理器类似，PowerPC 处理器使用堆栈实现函数的调用。PowerPC 处理器使用 GPR1 寄存器将整个堆栈段构成一个单向链表，在这个单向链表中的每一个数据成员被称为堆栈栈帧(Stack Frame)，其中每一个函数将对自己的栈帧进行维护。PowerPC 体系中的堆栈的增长方向是从高地址到低地址，而堆的增长方向也是从低地址到高地址。当栈段和堆相遇时就会产生溢出。PowerPC 体系的堆栈结构如图 2-5 所示。

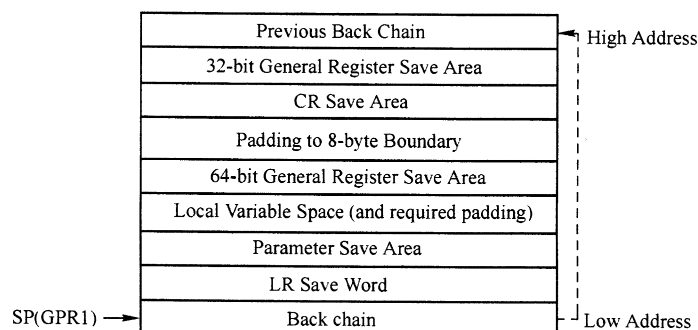


图 2-5 PowerPC 堆栈栈帧结构

在图 2-5 中，PowerPC 体系的每一个栈帧是由一个个数据成员组成的，其中有些数据成员间需要 8 字节对界。图 2-5 中各个数据成员的描述如下所示。

- Back Chain。当前的栈顶指针寄存器 SP(即寄存器 GPR1)保存上一个栈帧的 Back Chain 的地址。当函数返回时，SP 将指回上一个栈帧。
- LR Save Word。该数据成员用来存放使用 bl 指令进入相应函数后，LR 寄存器的值。当函数返回时，将从该数据成员中获取 LR 寄存器的值，然后再使用 blr 指令进行函数返回。
- Parameter Save Area。该数据成员用来存放函数的参数，E500 ABI 规定进行函数调用时，首先使用通用寄存器 GPRR3～GPR10 保存函数参数，如果函数的参数多于 8 个，剩余的部分参数将使用 Parameter Save area 保存。
- Local Variable Space。该数据成员用于存放函数的临时变量，E500 ABI 首先使用通用寄存器 GPR14～GPR31 存放函数的临时变量，如果一个函数使用的临时变量过多，则需要使用该数据成员存放临时变量。该数据成员与 64-bit General Register Save Area 间需要 8 字节对界，因此有时需要填充位。
- 64-bit General Register Save Area。该数据成员用于保存函数所使用的 64 位的寄存器。
- CR Save Area。该数据成员用于保存 CR 寄存器，如果函数没有使用 64 位的寄存器，那么 CR Save Area 和 64-bit General Register Save Area 之间的 padding 将被忽略。

- 32-bit General Register Save Area 用于保存函数所使用的 32 位寄存器。在一般情况下，E500 内核使用该数据成员保存处理器中的 32 位通用寄存器。但是如果在函数中使用通用寄存器中的高 32 位进行运算，如进行与 SPE 相关的运算，则需要使用数据成员 64-bit General Register Save Area 来保存这些寄存器。

如果程序员完全使用 C 语言进行程序设计，将不用考虑 PowerPC 体系的栈帧结构。因为此时整个栈帧结构将由 C 编译器维护。但是如果程序员需要在汇编程序中进行 C 语言函数调用或者在 C 程序中进行汇编语言函数调用时，则应注意栈帧的维护。

程序员在实现一个复杂的 C 程序调用汇编函数时，维护栈帧并不容易，稍不留神就会出错。这里提供一个相对较为简单的办法，首先使用 C 语言建立一个基本的函数，将所有的函数参数和即将在函数中使用的临时变量添加到这个函数中，然后使用 C 编译器生成汇编代码，此时生成的汇编代码将自动维护好 E500 内核的栈帧结构。

Linux PowerPC 中的一些汇编程序也是使用这种方法生成的。这里需要提醒读者注意，使用这种方法时，不要打开 C 编译器的优化选项，因为 C 编译器会删除这些暂时没有使用的临时变量。

多数在 C 程序中进行的汇编语言函数调用较为容易，例如在汇编语言中函数的参数少于 8 个，使用的临时变量用通用寄存器 GPR14～GPRR31 就足够存放，同时在这个汇编函数中不需要调用其他函数时，其栈帧结构就十分简单，如图 2-6 所示。

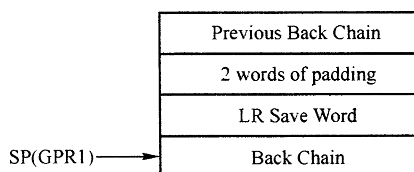


图 2-6　简单的 PowerPC 堆栈栈帧结构

在这种情况下堆栈中仅需要保存 Back Chain（用来维护 SP 指针）和 LR Save Word（用来进行函数返回）。对于此类栈帧结构，恐怕是刚刚入门的程序员也可以轻易掌握，并用来实现 C 程序调用汇编语言的函数调用。

这种栈帧结构也是 E500 内核可能的最小栈帧大小，共 16 个字节。Linux PowerPC 使用宏 STACK _'FRAME _ OVERHEAD 定义最小的堆栈栈帧的大小。这个宏的定义在 ./include/asm-powerpc/ptrace.h 文件中。

```
#define STACK_FRAME_OVERHEAD 16 /* size of minimum stack frame */
```

本节内容对于程序员深入理解 PowerPC 体系结构十分重要，指令集、寄存器、及与编译器有关的 ABI 是处理器体系结构的精华所在。由于篇幅所限，本书不能详述。

2.5　PowerPC 处理器的指令执行

PowerPC 处理器采用多发射（Superscaler）和乱序执行（Out-of-Order）技术实现指令的流水执行。所谓指令的流水执行是指将一条指令的执行全过程分解为若干个子任务，然后建立相关的流水线，将一个指令分为若干段分别执行。

处理器采用指令流水线的原因主要有两条，一是因为在现代处理器中在一个机器时钟周期之内无法完成从取值到指令完成的全过程，因此必须将指令执行分解为若干段；二是为了充分利用流水线各个功能部件。

指令的流水执行是处理器内核中的重要组成部件。基于不同内核的 PowerPC 处理器，采用的指令流水并不相同。如基于 604E 的处理器，其流水线长度为 6；而基于 E500 内核的处理器，其流水线长度为 7。有些处理器，如基于 Prescott 内核的奔腾，其流水线长度为 31，采用这种超长的流水线长度的主要优点是可以将一条指令的执行分解为更多的步骤，这样可以进一步细化指令的执行单元，从而可以使处理器运行在更高的主频上。

但是，有时这种主频的提高并不能提高处理器指令的执行效率。如果一个处理器使用时间过长，那么在程序分支预测失败后，指令流水线的排空时间也随之加长，从而极大地影响了流水线的建立时间。因此虽然加大流水线的深度可以有效地提高处理器的主频，但是片面地提高流水线的深度并不能有效地提高处理器运算性能。

E500 内核使用的指令流水线深度为 7 级，如图 2-7 所示。

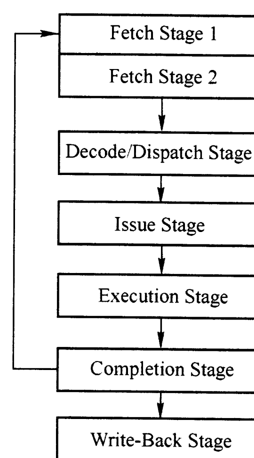


图 2-7　E500 内核的指令流水线

- 指令预取（Instruction Fetch）。E500 内核使用两级流水线实现指令的预取，首先从 L1 指令 Cache 中获得需要执行的指令；如果所取的指令在 L1 Cache 没有命中，E500 内核将依次访问 L2 Cache 和外部存储器。
- 指令译码（Instruction Decode/Dispatch）。指令译码模块将根据指令的 OPCD 码，将指令和与此指令有关的操作数进行分解，分别放入指令译码模块的缓存中。
- 指令发射（Instruction Issue）。指令发射模块根据指令执行模块的使用情况，决定是否将指令发射到指令执行模块中执行。
- 指令执行（Instruction Execution）。指令执行模块将执行指令。E500 内核中一共有 BU，L/SU，SU1，SU2 和 MU 共 5 个执行单元。其中 BU(Branch Unit)用来执行转移指令和对 CR 寄存器操作的指令，L/SU（Load/Store Unit）用来执行存储器访问指令，SU1/SU2 用来执行简单的算术逻辑运算，而 MU（Multiple-cycle Instruction Unit）用来执行一些需要多个机器周期的指令。
- 指令完成（Instruction Completion）。该模块将完成指令的执行，将执行结果保存到相应的寄存器中，同时释放相应指令所占用的流水线资源。
- 指令回写（Instruction Write-Back）。同步 Rename Register（重命名寄存器）与系统通用寄存器，有些指令的执行不经过此流水部件。

下文将指令从预取到结束的全过程称为指令运行，而将指令在指令执行部件中运行称为指令执行。E500 内核在指令运行的各个环节之间设置了许多缓冲，并在指令发射部件中使用重命名寄存器实现程序的乱序执行。E500 内核采用 7 级流水结构运行指令，在一个机器周期内可以发射两条指令，并可以完成两条指令的运行。

E500 内核的乱序执行是指在指令的执行阶段，由于不同指令的执行速度不同或者由于其他原因，后执行的指令可以提早执行完毕。为此在 E500 内核中设立了一些重命名寄存器（Renaming Register）用来实现这一技术。但是在 E500 内核中，指令在发射阶段和完成阶段必

须顺序进行。

2.5.1 指令预取

E500 内核使用两个阶段实现指令的预取。在指令预取的第一阶段,E500 内核首先从 L1 Cache 中获取这些指令。如果 L1 Cache 没有命中,E500 内核将通过 ILFB(Instruction Line Fill Buffer)从系统总线上的 L2 Cache 或者主存储器中获取指令,ILFB 一次可以存放 8 条指令。对禁止 Cache 的空间进行指令预取时必须使用 ILFB 进行缓冲。

通过各种方式预取的指令将被 E500 内核存放到指令队列 IQ(Instruction Queue)中,该指令队列的长度为 12。指令预取的第一阶段如图 2-8 所示。图中虚线表示控制总线通路而实线表示数据总线通路。现代处理器系统中将控制通路与数据通路分离,这也是处理器内核中常用的方法。

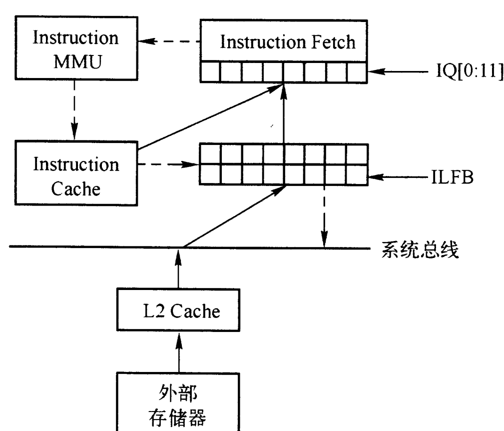


图 2-8　E500 内核的指令预取的第一阶段

在 E500 内核中,只要 IQ 队列不为空,指令预取单元就会在每个机器周期内将指令源源不断地送往 IQ 队列。

指令预取的第二阶段首先处理在 IQ 队列头部的两条指令,即存放在 IQ0 和 IQ1 中的指令。具体地说,指令预取的第二阶段将根据指令编码 OPCD 的不同将指令分为两类,跳转指令和通用指令。指令的译码与发射单元会将这些指令分别加入到转移发射队列 BIQ(Branch Issue Queue)或者两条通用发射队列 GIQ(General Issue Queue)中,然后再进入相应的指令执行单元。在指令预取的第二阶段所做的指令编码的预处理将减轻指令译码单元的负担。

PowerPC 体系结构的指令长度为 32 位。E500 内核在每个机器时钟内最多可以预取 4 条指令,而且这 4 条指令不能跨越指令 Cache 行边界。

由上文可知,E500 内核指令的预取使用两个机器时钟,第一个机器时钟用作预取指令,第二个机器时钟用来分析指令,然后将不同指令加入到相应的指令队列中。此外,在第二个机器周期中还需要对转移指令进行处理。E500 内核不支持转移指令的静态分支预测,而是在指令预取的第二阶段开始考虑如何处理程序的分支预测。E500 内核程序预测的处理较为复杂。下文将详细介绍程序的分支预测。

指令预取是指令运行过程中相对较为简单的部件。如果忽略程序分支预测带来的时间损

失，指令的预取部件的执行效率非常高。不过在以下几种特殊情况下，指令预取效率将会受到较大的影响。

- 对 L1 Cache，L2 Cache 及 BTB(Branch Target Buffer)进行管理的指令在执行时将阻塞指令的预取。
- 指令 MMU 和指令 L1 Cache 不命中时将会阻塞指令的预取，直到 ILFB 从 L2 Cache 或者外部存储器中获得指令。
- 在指令预取的第二阶段，如果进行指令转移时，E500 内核将把在第一阶段预取的指令丢弃，此时 E500 内核将重新进行指令预取。

IQ 队列满时，将停止指令的预取。

2.5.2 指令译码与发射单元

E500 内核的指令译码单元在一个机器时钟内可以处理两条指令的译码，然后将这两条指令放入到指令队列 BIQ 或者 GIQ 中。指令译码与发射单元的结构如图 2-9 所示。

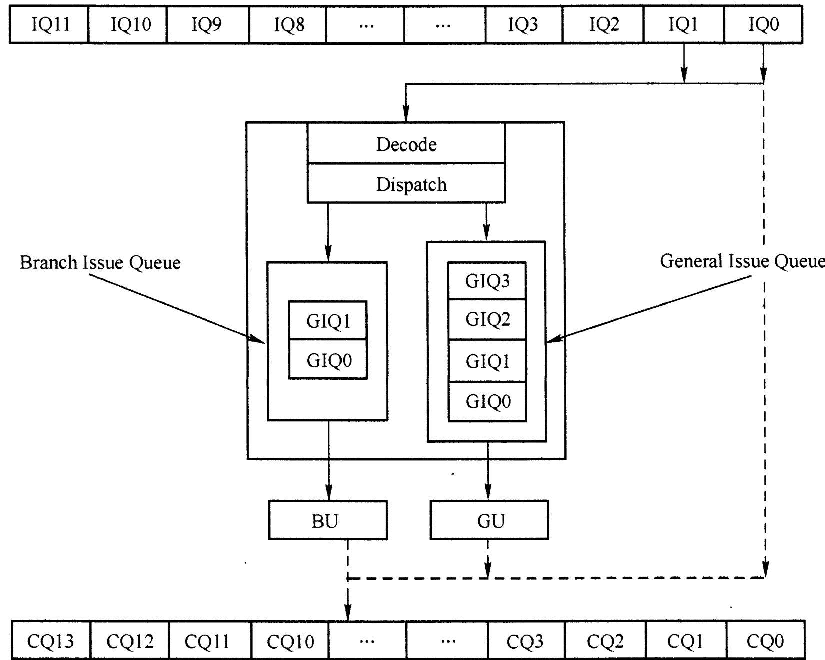


图 2-9　E500 内核的指令译码与发射单元结构图

指令译码单元将对 IQ 中的指令顺序进行译码，并将这些指令派遣到相应的指令发射部件中。一些特殊的指令如“isync”，“rfi”和“nop”指令将由指令译码单元直接传递到指令执行单元而忽略指令发射单元。

- 指令译码单元将对 IQ0 和 IQ1 中的指令进行译码，如果 CQ(Completion Queue)队列不为空，则将 IQ0 和 IQ1 中的指令派遣到 BIQ 或者 GIQ 队列中，同时改变 CQ 队列的队首和队尾指针。
- BIQ 的队列深度为 2，GIQ 的队列深度为 4。在一个机器周期内，BIQ 队列一次可以接受一条分支转移指令而 GIQ 一次可以接受两条非分支转移类指令。这些指令只能来

自 IQ 队列头部的两条指令,IQ0 和 IQ1。

当 IQ0 和 IQ1 中存放的指令为无条件转移指令时,指令译码部件将阻止下一条指令的译码直到此无条件转移指令执行后。为提高效率,E500 内核不需要将无条件转移指令完全执行完毕,才进行下一条指令的译码。E500 内核在确定无条件转移指令的执行地址后,将立即重新进行指令预取,之后重新对新的指令进行译码。

E500 内核为支持指令的乱序执行,在指令译码阶段,将从 E500 内核中获得一些相关的 Rename Registers,以便在指令执行单元中使用。

正常情况下,指令译码单元可以在每个机器时钟节拍中对两条指令进行译码。但在以下几种特殊情况下,指令译码单元将会被阻塞。

- 有些指令需要进行指令同步。此时 IQ 队列中的指令需要等待指令同步结束后才能被译码。
- 有时当前指令的执行需要等待指令流水线的排空。
- 当动态指令预测失败时。此时的处理比较复杂。指令译码在此时将会被阻塞,直到处理完因为预测失败而带来的一系列问题。
- 当指令完成队列 CQ 中没有空间时,指令发射单元单元将被阻塞,下文将在指令完成单元中详细介绍 CQ 的使用。
- 在 IQ 队列中没有指令时。此时显然不能进行指令译码,发生这种情况的主要原因是预取的指令没有在指令的 L1 Cache 中命中。
- 当 BIQ 和 GIQ 满时,而且执行单元忙时。此时不能进行指令译码。
- 其他因为资源相关所带来的阻塞。

指令译码结束后,IQ0 或者 IQ1 中的指令将被分别放入 BIQ 或 GIQ 队列中等待执行。E500 内核支持多发射技术,在一个机器时钟节拍中,可以同时发射多条指令,其中所有的分支转移指令将由 BIQ0 传递给指令执行单元,非分支转移指令将由 GIQ0 和 GIQ1 传递给指令执行单元。

指令发射单元支持乱序处理。当 GIQ0 中的指令由于资源问题没有被发射,而 GIQ1 中的指令被发射时,GIQ2 中的指令可以下移到 GIQ1 中继续发射。指令发射单元还会将一些组合指令如 stwu,lmw 等指令分解成为简单的单操作指令,然后再将这些指令分别发射到对应的执行单元中,如 stwu 指令将被分解为 stw 和 addi 两条指令,然后分别被发射到 L/SU 和 SU1(SU2)中。

由此可见,stwu 指令虽然可以同时实现访问存储器和将地址更新两种操作,可是在指令运行时还是要分为两个步骤进行,但是此指令的运行还是比分解为 stw 和 addi 两条指令执行的效率高,因为 stwu 指令在指令预取和译码部件中仅执行一次,采用这种指令可以减小可执行程序代码的大小,同时提高指令发射的效率。

正常情况下,指令发射单元可以在每个机器时钟节拍中发射一条指令到 BU 单元,两条指令到其他指令执行单元中。但在以下几种特殊情况下,指令发射单元将会被阻塞。

- 在 BIQ 或者 GIQ 队列中没有指令。
- 指令发射单元需要等待运行资源,如指令执行单元运算资源没有被释放,或者由于指令的相关性,指令发射单元需要等待前一条指令执行完毕之后,才能够将在 GIQ 队列中的指令发射到指令执行部件中。

2.5.3 指令执行单元

E500 内核支持乱序执行并支持多发射。为此 E500 内核采用了寄存器重命名技术并设置了多个指令处理单元。

在指令发射单元和指令执行单元之间有一些预约站(Reservation Station),在指令从 BIQ 或 GIQ 中进入执行单元之前,这些指令需要对预约站的状态进行检查。如果预约站忙,在指令发射单元将被阻塞,如果不忙,这些指令将进一步做相关性检查,如果没有相关性发生,这些指令将同时进入预约站和指令执行单元。

E500 内核采用分支处理单元 BU(Branch Unit)对转移指令进行动态预测,使用了 SU1 (Simple Unit),SU2,一个 LSU(Load/Store Unit)和一个 MU(Multiple-cycle Instruction Unit)用于指令的执行。因此 E500 内核最多一次可以执行 5 条指令。指令执行单元是指令运行的 7 个阶段中最为复杂的单元,其结构如图 2-10 所示。

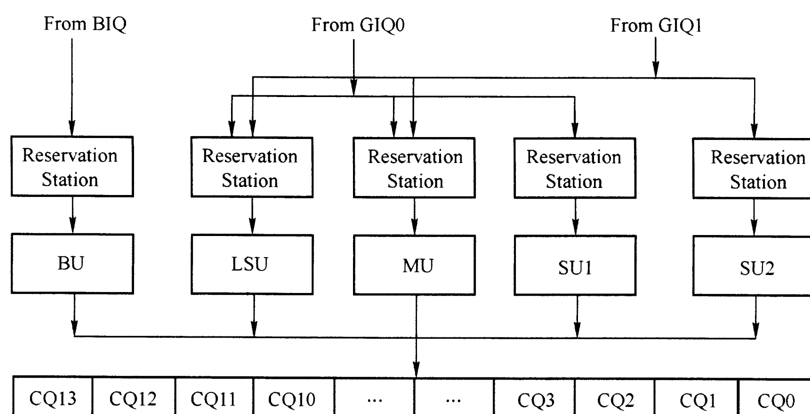


图 2-10 E500 内核的指令执行单元结构图

指令执行单元通过预约站与 BIQ 和 GIQ 进行相联,指令在执行过程中将会从相应的通用寄存器,或重命名寄存器中获取操作数,然后执行。其中 BIQ0 可以与 BU 进行数据传输, GIQ0 可以与 SU1,LSU 与 MU 进行数据传输,GIQ1 可以与 SU2,LSU 与 MU 进行数据传输。

1. SU 与 MU 处理单元

E500 内核具有两个 SU 处理单元,SU1 和 SU2。其中 SU1 可以执行有关 32 位,64 位的定点或者浮点的算术及逻辑运算,SU1 中还可以执行 SPE 中的指令。而在 SU2 中只能执行一部分 SU1 中的指令,64 位的 SPE 运算不能在 SU2 中执行。

在 SU 处理单元中执行的大部分指令只需要一个机器周期,但是有些访问某些 spr 寄存器的指令可能会需要多个机器周期。

MU 处理单元用来执行乘除运算,在 MU 中执行的乘法操作一般需要 4 个机器时钟节拍。而除法将根据运算的复杂性可以在 4 个、11 个、19 个或 35 个机器时钟内结束。SU 与 MU 是处理单元中较为简单的模块。

2. LSU 处理单元

LSU 处理单元用来执行存储器访问指令及管理 Cache 和 TLB 的一些指令。与其他指令

处理单元相比，LSU 较为复杂。在 LSU 处理单元中执行的指令首先需要通过 E500 内核的 MMU 计算被访问数据的物理地址，之后使用 E500 内核的内存系统，依次经过 L1，L2 Cache 和外部存储器，对数据进行访问。

LSU 还需要处理存储器访问的同步与相关。如果在存储指令执行时，没有出现存储访问异常，如 Cache 不命中、存储相关、存储同步等问题，在 LSU 中执行指令的需要 3 个机器时钟节拍，为此在 LSU 共分三级流水处理在此运行的指令。

E500 内核的 LSU 中采用了 L1 Store Queue，LMQ(L1 Load Miss Queue)，DLFB(Data Line Fill Buffer)和 DWB(Data Write Buffer)等数据缓冲，来缓解主存储器的瓶颈问题，LSU 的结构如图 2-11 所示。

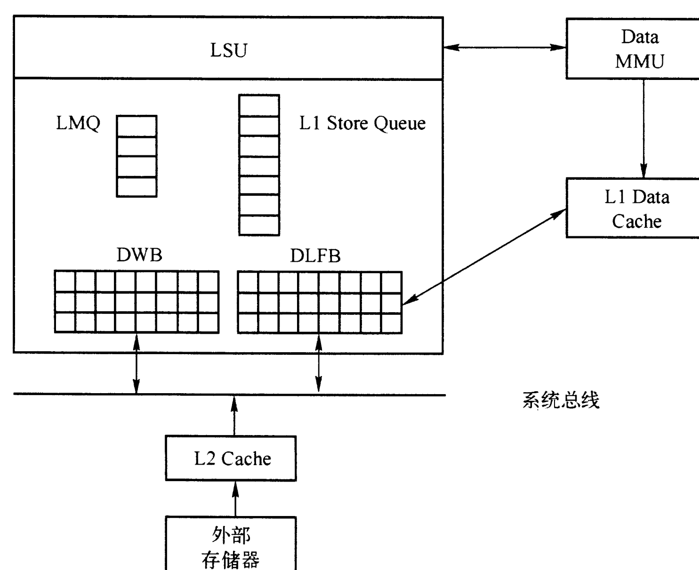


图 2-11　E500 内核 LSU 处理单元

(1) L1 Store Queue。执行存储器写指令时，LSU 首先将存储器写指令放入 Store Queue 队列中，同时对存储器进行写操作。当存储器写指令执行完毕后，Store Queue 中的对应存储器写指令将被移出。E500 内核的 Store Queue 深度为 7，最多可以支持 7 条存储器写指令。当 Store Queue 满时，存储器写指令将被阻塞。

(2) DLFB。DLFB 用来缓存 L1 Cache 的数据，DLFB 可以理解为 L1 Cache 到系统总线上存储部件的缓冲。进行存储器读写操作时，如果 L1 Cache Miss，处理器将通过 DLFB 从系统总线中获取数据。E500 V1 内核中的 DLFB 的宽度为 256 位，深度为 3。

(3) LMQ。执行存储器读指令时，LSU 首先访问 L1 Cache，如果数据在 L1 Cache 中命中，则获得此数据；反之，LSU 将为这条存储器读指令分别在 LMQ 和 DLFB 中各分配一个 Entry。其中 LMQ 的 Entry 中存放读指令，DLFB 用来存放即将读取的数据。

LMQ 和 DLFB 准备好后，LSU 发起总线请求从系统总线中获取数据。当存储器读指令执行完毕后，LMQ 中的 Entry 将被移出。在 E500 V1 内核中，LMQ 中用来存放存储器读指令，深度为 4。

(4) DWB。DWB 用来暂存失效的 Cache Line。当 L1 Cache 需要替换更新时，DWB 用来存放失效的 Cache Line 并选择在合适的时机将 L1 Cache 与存储器系统进行同步。E500 V1 内核的 DWB 宽度为 256 位，深度为 3。

3. BU 与 BPU 处理单元

BU 单元与 BPU(Branch Prediction Unit)不同，BU 用来处理转移指令和对 CR 寄存器操作的指令，而 BPU 用来判断转移指令是否发生转变。

BPU 虽然与指令执行单元有千丝万缕的关系，但是它不完全属于指令执行单元，BPU 在指令的译码阶段就开始对分支转移是否发生转移进行预测。

E500 内核中的 BU 处理单元使用两级流水线对分支转移指令进行处理。分支转移指令将分为分支转移指令执行 BE(Branch Execution)和分支转移指令执行完成 BF(Branch Finish)两个处理阶段。其中，每个阶段占用一个机器周期。E500 内核采用专用通道技术处理 BU 处理单元的两级流水线。例如 crand 指令在 BF 中执行时，可以通过专用通道将 CR 寄存器中的变化转给 BE 中与 CR 寄存器相关的指令，从而提高流水线的效率。

如果转移预测成功，BU 处理单元运行十分顺利，不会对指令流水线造成任何破坏；如果转移预测失败，BU 处理单元将重新把指令预取到 IQ 队列后，重新启动指令流水线，这种情况将极大地影响指令流水线的执行效率。

E500 内核不支持静态分支预测，而是采用 BPU 实现动态分支预测。在 E500 内核内，BPU 中的 BTB(Branch Target Buffer)共含有 512 个 Entry，这 512 个 Entry 使用 4 路组相联结构进行管理。BTB 的 Entry 中共有 4 个状态：

- Strongly not taken(不进行转移处理，00)。
- Weakly not taken(建议不进行转移处理，01)。
- Weakly taken(建议进行转移处理，10)。
- Strongly taken(建议进行转移处理，11)。

在 E500 内核复位后，BTB 中所有的 Entry 状态为 Invalid，即当前 Entry 无效。当一条转移指令首次进入指令队列中，指令译码单元不会为此指令分配 BTB 中的 Entry，此时分支转移指令采用缺省的 Strongly not taken 的状态进行。

如果该指令没有进行转移，那么 BPU 将不会为该指令在 BTB 中分配一个 Entry；如果该指令进行转移，BPU 将在 BTB 中分配一个 Entry 用来记录本条转移指令，此时该 Entry 被设置为有效，其状态为 Srongly taken(11)；当该转移指令第二次执行时没有转移时，将 Entry 中状态的值减 1，即为 Weakly taken(10)；再次没有转移时，将状态的值继续减 1，即为 Weakly not taken(01)；如果该转移指令发生转移时，将 Entry 中状态加 1，并以此类推。

E500 内核中可以禁止这种动态分支预测功能，E500 内核提供了 BBEAR 和 BBTAR 两个寄存器和 bblels 和 bbelr 两条指令用来锁定或者解除锁定对某一条指令的分支预测功能。指令译码单元将参考 BPU 中转移指令的状态，决定如何调整指令流水，如何进行指令预取等。

4. 指令完成单元与指令回写单元

指令完成单元使用了指令完成队列 CQ(Completion Queue)用来结束指令的运行。在 E500 内核中，CQ 的深度为 14，这也意味着在 E500 内核中，指定的一个机器周期内，最多有 14 条指令在同时运行。E500 内核规定 CQ 中的指令根据指令译码的顺序依次完成。

E500 内核在一个机器周期，最多可以有两条指令执行完毕。这两条指令存放在 CQ0 和

CQ1 中。在 E500 内核中,有些指令只能在 CQ0 中完成。如果 CQ0 中的指令因为各种情况不能完成时,CQ1 中的指令也不能完成。E500 内核的这种处理方法保证了指令还是按照指令译码的顺序完成的。

在 E500 内核中,CQ 队列中的 Entry 由指令译码单元进行初始化,在指令译码阶段,将从 CQ 队列中一次取出两个未用的 Entry,并将此 Entry 的状态置为未完成,然后通过指令发射单元送入指令执行单元,进行指令执行,在执行完毕后将 CQ 队列中相应 Entry 的状态置为已完成。最后在指令完成单元中将 CQ 队列中的对应 Entry 删除,从而完成指令的运行。

值得注意的是本节出现的 IQ,CQ 队列并不是只在相应的执行单元中使用。对于 IC 设计工程师而言,IQ 和 CQ 队列不过是设计指令运行全过程中使用的一些缓冲区和相应的控制位,在 IC 设计中 IQ 和 CQ 队列的使用贯穿指令运行的各个单元。

在指令完成单元中值得注意的是,当分支转移指令预测失败后,指令完成单元将负责将流水线清除,之后开始对指令进行预取。E500 内核在指令完成单元中集中处理各种同步问题。这些同步问题包括各种指令数据同步,中断及异常事件等。

指令回写单元较为简单,在每个机器时钟周期内,指令回写单元将使用的重命名寄存器写入相应的 GPR 寄存器和 CR 寄存器中,然后释放重命名寄存器的空间。指令回写单元的执行将不会被阻塞。

从 E500 内核的指令运行的全过程中,我们可以发现除了在指令回写单元外,指令在其运行的各个环节中都有可能被阻塞。如何充分利用指令流水线,是一个从系统到应用各个方面都需要考虑的问题。

2.6 E500 内核的乱序执行

E500 内核支持指令的乱序执行,乱序执行是指处理器可以将指令不按程序规定的顺序发送给各个指令执行单元。当指令执行完毕后,E500 内核再将运算结果重新按程序指定的指令顺序排列。

采用乱序执行技术的目的是处理器不需要等待一条执行速度较慢的指令执行完毕,就可以执行下一条指令。从而有效地提高了处理器各个运算单元的使用效率与处理器的运算速度。

现代处理器大都采用多发射流水线技术,因此在指令运行阶段会遇到各种同步事件,这些同步事件有时也会降低处理器乱序执行的效率。由于多发射流水的存在,指令乱序执行的设计较为复杂,需要与指令运行中许多单元进行配合,因此有些高级处理器在设计中并没有实现乱序执行。

2.6.1 指令乱序执行的例子

下文首先假定在一个不支持乱序执行的处理器 A 中执行(R2 * R4) + R2 * (R2 + R2)这样的运算,其中寄存器 R2 中存放无符号数 UIMM,寄存器 R4 中使用寄存器 R1 进行间接寻址,获得相应的数据。其汇编程序如表 2-3 所示。

表 2-3 (R2 * R4) + R2 * (R2+R2)的汇编程序

指令顺序	指　令	指令描述	指令延时
I1	Load R2, UIMM	R2 ← UIMM	1
I2	Load R4, 34(R1)	R4 ← mem(R1)	4
I3	Mult R6, R4, R2	R6 ← R2 * R4	3
I4	Add R8, R2, R2	R8 ← R2 + R2	1
I5	Mult R4, R2, R8	R4 ← R2 * R8	3
I6	Add R10, R6, R4	R10 ← R6 + R4	1

假定在处理器 A 中,乘法指令执行时使用 3 个机器周期,而存取存储器数据指令执行时使用 4 个机器周期,其他的指令执行时使用一个机器周期。为简化起见,假定处理器 A 不支持多发射,但是支持指令流水,同时拥有多个指令处理单元。

由此可见,处理器 A 在一个机器周期内最多只能发射一条指令,最多只有一条指令可以执行完毕。在这种情况下,I1～I6 指令的执行顺序将如表 2-4 所示。在 2-4 表中“Clock”是指处理器使用的机器周期,假定 I1 在机器周期 1 处开始执行;“顺序执行”一栏中,列出指令在哪个机器周期开始执行,同时指出指令流水停滞的原因;而“执行结束”一栏中,列出哪个指令在该机器周期中执行完毕。

表 2-4 (R2 * R4) + R2 * (R2+R2)的顺序执行过程

Clock	1	2	3	4	5	6	7	8
顺序执行	I1	I2	I3 等待操作数 R4				I3	I4
执行结束		I1				I2		
Clock	9	10	11	12	13	14	15	16
顺序执行	I5 等待操作数 R8		I5	I6 等待操作数 R4			I6	
执行结束	I4	I3				I5		I6

- 指令 I1 和 I2 可以立即执行,但是由于处理器 A 只支持单发射,所以 I2 必须在机器周期 2 中开始执行。
- 指令 I3 由于使用了寄存器 R4,而该寄存器在指令 I2 执行完毕后才被释放,因此指令 I3 只能在机器周期 7 中开始执行,因为在机器周期 6 中指令 I2 执行完毕,指令 I4 没有任何相关性问题,可以在机器周期 8 中开始执行。
- 指令 I5 使用了寄存器 R8,该寄存器在指令 I4 执行完毕后才被释放;因此该指令只能在机器周期 11 中开始执行。
- 指令 I6 使用了寄存器 R4,该寄存器在指令 I5 执行完毕后才被释放,因此该指令只能在机器周期 15 中开始执行。
- 最后在机器周期 16 中,结束指令 I6 的执行。

通过以上分析可知,由于有些指令的相关性不可避免,处理器 A 的流水线并没有被充分利用。指令 I1～I6 在顺序执行计算(R2 * R4) + R2 * (R2 + R2)时一共需 16 个机器周期。

通过对以上指令进行分析,可以发现 I4 指令所需的操作数只有 R2,而且 R2 在 I1 指令执行

完毕后就已经被准备好,从而执行 I4 指令不会和其他指令产生相关。因此如果采用乱序执行机制,I4 将可以被提前执行。但是采用这种乱序执行技术就一定能够提高指令执行的效率吗?

为此假定处理器 B 继承了处理器 A 的一些特性,但是支持指令的乱序执行。在这种情况下,I1～I6 指令的在处理器 B 执行的顺序将如表 2-5 所示。

表 2-5 (R2 * R4)+ R2 * (R2+R2)的乱序执行过程

Clock	1	2	3	4	5	6	7	8
乱序执行	I1	I2	I4	I3 等待操作数 R4,I5 等待 I2 中使用的资源释放			I3	
执行结束		I1		I4		I2		

Clock	9	10	11	12	13	14	15	16
乱序执行	I5 等待 I3 中使用的 R4 释放		I5	I6 等待操作数 R4			I6	
执行结束		I3				I5		I6

可以发现,虽然处理器 B 采用了乱序执行机制,但是流水线还是没有被充分利用。使用处理器 B 计算(R2 * R4) + R2 * (R2 + R2)仍然需要 16 个机器周期。这个执行结果令人失望,因为采用指令乱序执行机制并没有改进程序的运行效率。程序的相关性依旧对程序的乱序执行机制产生影响。我们需要进一步分析:究竟是什么样的相关影响了指令乱序执行的效率?

2.6.2 指令的相关性

指令运行时,有时因为未执行完毕的指令占用了一些资源,从而造成后面的指令无法立即运行,必须等待前面的指令执行完毕,这种情况称为指令的相关。

在程序执行过程中存在许多种相关,有些相关是无法避免的,而有些相关可以通过某种机制规避。程序执行中经常出现的相关主要有以下几种。

(1) 控制相关。此类相关问题由程序的转移指令引起。转移指令的下一条指令要等待转移指令的结果决定是否执行。在 E500 内核中使用 BPU 和 BU 处理单元处理所有的转移类指令并对控制相关进行处理。

(2) 资源相关。此类相关是指当前指令所需的资源缺乏而引起的相关。比如当前指令需要使用 MU 处理单元,但是 MU 处理单元正在被其他指令所使用,此时当前指令必须要等待其他指令执行完毕,释放 MU 处理单元后,才能被执行。

(3) 数据相关。数据相关是指在程序的执行过程中,下一条指令需要等待前一条指令的执行结果,如下所示。

```
RAW-I1:    mul  r2, …, … ;
RAW-I2:    add  …, r2 , … ;
```

此时 RAW-I2 需要等待 RAW-I1 指令执行完毕,获得寄存器 r2 的值后才能继续执行。这类相关也称为 Read-after-Write 相关(Read-after-Write dependencies),简称为 RAW 相关。

(4) 存储相关。产生这类相关的主要原因是不同指令在处理器的流水线中运行时,不同的指令所需要的机器周期不同,后执行的指令有可能影响前一条指令的执行结果。此类相关分为两类:Write-after-Read 相关(Write-after-Read dependencies),简称为 WAR;Write-after-

Write 相关(Write-after-Write dependencies),简称为 WAW。

WAR 和 WAW 相关的例子如下所示。

WAR-I1:　mul　…, r2 , … ;

WAR-I2:　add　r2, …, … ;

由于指令的延时不同,虽然指令 WAR-I1 将先于 WAR-I2 进行译码和发射,但是指令 WAR-I2 仍然会被提前执行完毕。此时指令 WAR-I2 将对寄存器 r2 进行修改,而这种修改必定会造成指令 WAR-I1 执行错误。此类相关问题被称为 WAR 相关。

WAW-I1:　load　r2, …, … ;

WAW-I2:　add　r2, …, … ;

假如指令 WAW-I1 将从外部存储器中取出数据放到寄存器 r2 中,但是由于取数操作慢于指令 WAW-I2 中的加法操作,因此指令 WAWI2 将提前执行完毕,提前修改寄存器 r2,导致了指令 WAW-I1 的执行错误。此类相关问题被称为 WAW 相关。

由于发生 WAR 和 WAW 相关的原因是在后面的指令执行对在之前的指令执行造成影响,因此此类相关也称为反向相关。

一些高级的处理器采取了各种办法解决以上提到的各种相关问题。比如,为了减轻控制相关对程序执行效率的影响,有些处理器可以对指令进行静态和动态分支预测。

不过这些静态和动态分支预测处理面对复杂的 if-else 或 switch-case 的多重嵌套,还是显得很无奈。程序员根据所使用的处理器结构而调整程序也许是一个较为合理的解决方案。目前,各类处理器在激烈的竞争中,结构日益趋同,一个在奔腾处理器下运行效率较低的程序在其他处理器下运行结果也不会有质的提高,估计其间的差别只是从特别差到比较差而已。

目前没有太好的办法解决数据相关。数据相关是由指令的执行顺序导致的,不可避免。程序员可以对算法进行优化改进,去减缓或者规避数据相关。而使用寄存器重命名技术是解决存储相关的有效方法。采用寄存器重命名技术可以成功地消除 WAR 和 WAW 相关。目前支持乱序执行的处理器基本上都采用了寄存器重命名机制。

2.6.3 寄存器重命名机制

寄存器重命名的概念并不复杂,就是将指令运行时所使用的寄存器用重命名寄存器进行替换,之后再将重命名寄存器进行回写的技术。采用此技术后,需要将上文出现 WAR 相关的例子中 I5 指令使用的 R4 寄存器改为 R4′,I6 指令中使用的 R4 寄存器也要使用 R4′,如表 2-6 所示。

表 2-6　使用重命名寄存器技术后(R2 * R4)＋ R2 * (R2＋R2)的乱序执行过程

指令顺序	指　令	指令描述	指令延时
I1	Load R2, UIMM	R2 ← UIMM	1
I2	Load R4, 34(R1)	R4 ← mem(R1)	4
I3	Mult R6, R4, R2	R6 ← R2 * R4	3
I4	Add R8, R2, R2	R8 ← R2 + R2	1
I5	Mult R4′, R2, R8	R4′ ← R2 * R8	3
I6	Add R10, R6, R4′	R10 ← R6 + R4′	1

采用重命名寄存器技术后，寄存器 R4 将被重命名为 R4′，此时处理器再采用乱序执行的方法就可以提高程序的效率。

由表 2-7 可知，寄存器重命名方法成功的解决了 I5 和 I3 之间的 WAR 相关，从而指令 I5 可以提前到 I3 指令之前运行，提高了程序的运行效率。

表 2-7　使用重命名寄存器技术后(R2 * R4)＋ R2 * (R2＋R2)的乱序执行过程

Clock	1	2	3	4	5	6
乱序执行	I1	I2	I4		I5	
执行结束		I1		I4		I2
Clock	7	8	9	10	11	12
乱序执行	13		I6 等待操作数 R4		I6	
执行结束		I5		I3		I6

这里需要提醒读者注意，在表 2-7 的指令乱序执行结束后，指令的运行仍然需要按照程序规定的顺序完成。

尽管重命名寄存器的概念比较简单，但是如何管理重命名寄存器，何时才能将存放在重命名寄存器中的数据回写到处理器的寄存器中，如何保证指令间的同步，及一共需要设置多少个重命名寄存器等一系列问题，对于处理器指令设计而言并不是一件十分容易的事情。不同的处理器在实现重命名寄存器机制时采取了不同的方法。

1. E500 内核的重命名寄存器

不同的处理器采用不同的方法将重命名寄存器放置在不同的位置。处理器一般使用以下几种方法存放重命名寄存器。

- 采用重命名寄存器与系统寄存器(在 PowerPC 处理器中，这些系统寄存器包括通用寄存器 R0～R31 和 CR 寄存器)独立的结构。如上文中提到将系统寄存器 R4 首先改为重命名寄存器 R4′，然后再将重命名寄存器 R4′回写到系统寄存器的做法，就是采用了这种结构存放重命名寄存器。
- 采用重命名寄存器与系统寄存器合并的结构。采用此方法的最大优点是不需要将重命名寄存器中的数据重新回写到系统寄存器，从而稍微提高了指令运行的效率。但是采用这些方法会稍微加大对系统寄存器的访问延时。DEC 的 Alpha21264 处理器采用了这种方法存放重命名寄存器。
- 重命名寄存器的索引存放在指令完成队列 CQ 中。此时处理器在发射指令之前将严格按照指令执行序列分配 CQ，在指令的执行过程中 CQ 将根据指令的执行状态对重命名寄存器进行管理，并要保证指令一定按照程序执行的顺序完成。当指令运行结束后 CQ 中相应的 Entry 被释放。E500 内核有可能采用了这种结构。
- 采用指令动态发射的处理器可以将重命名寄存器放在指令发射单元和指令执行单元之间的数据缓冲中，这个数据缓冲一般被称为 shelving buffer。采用指令动态发射的处理器共分两个步骤进行指令发射。首先将要发射的指令放入缓冲，然后对放入缓冲中的

数据进行相关性检查，之后再决定指令是否执行。

在一个处理器内核的设计中，采取何种方法管理系统的重命名寄存器涉及许多具体的量化计算，上文提到的管理重命名寄存器的方法都在具体的处理器中得到了应用，这些方法各有优劣。

Book E 内核和 E500 内核手册并没有提供有关重命名寄存器的详细设计资料，不过我们仍然可以从已有的资料中获得一些有用的信息。E500 内核在指令的译码阶段为每条指令分配一个重命名系统寄存器和一个重命名 CR 字段寄存器，尽管有些指令可能并不需要重命名寄存器。为此 E500 内核提供了 14 个重命名寄存器和 14 个 CRn 寄存器。

重命名寄存器和 CQ 队列保证了指令可以在执行阶段乱序而在指令运行结束后按程序的顺序结束，并将重命名寄存器写入相应的系统寄存器中。

这里要提醒读者注意，本书作者并没有获得 E500 内核有关寄存器重命名设计的详细资料。因此以下文字是有关寄存器重命名技术的一种实现方式，并不作为 E500 内核的参考。下文将用处理器 C 对 E500 内核进行替换，其中处理器 C 具有除了寄存器重命名实现机制外 E500 内核的一切特性。

在实现寄存器重命名结构时，处理器 C 为系统寄存器在指令运行时维护两张表。其中一张表用来记录系统寄存器和重命名寄存器的关系，另一张表用来描述重命名寄存器的结构，本文将这两个表分别称为重命名寄存器索引表和重命名寄存器描述表。

2. 重命名寄存器的索引表与描述表

重命名寄存器的索引表见表 2-8。

处理器 C 共支持 32 个通用寄存器，因此此索引表的深度为 32，即为每一个通用寄存器建立与重命名寄存器相关的索引，该索引表由三项组成。

表 2-8 重命名寄存器索引表

Number	Valid	Rename Register Index
0	1	12
1	1	3
	…	
10	1	7
11	1	8
12	1	9(13)
	…	
31	1	10

- Number 字段。该字段用来记录 32 个通用寄存器的号码。在 IC 设计中，可以用重命名寄存器索引表的号码替代该字段，以节约资源。
- Valid 位。该位用来表示当前通用寄存器是否被重命名。为 1 时表示该通用寄存器已经被重命名，为 0 表示该通用寄存器没有被重命名。
- Rename Register Index 字段。该字段用来表示重命名寄存器和当前的通用寄存器对应关系。

在处理器 C 中一共有 14 个重命名寄存器。因此最多能够建立 14 个对应关系。如表 2-8 所示，寄存器 r10 所对应的 Entry 的 Valid 位为 1，Rename Register Index 为 7，此 Entry 表示重命名寄存器 7 与 R10 相对应。

重命名寄存器的描述表见表 2-9。

表 2-9　重命名寄存器描述表

Number	Valid	Dest Reg Number	Latest bit	Value	Value Valid
0	0				
1	0				
…					
7	1	10	1	15	1
8	1	11	1	9	1
9	1	12	0	50	1
…					
13	1	12	1	20	1

处理器 C 一共有 14 个重命名寄存器，因此重命名寄存器描述表的深度为 14。该表的每一 Entry 由 5 个字段组成。

- Valid。此位用来表示当前 Entry 是否有效，为 0 时表示此 Entry 无效可以被重命名寄存器索引表使用，为 1 时表示该表项已经与重命名索引表建立连接。
- Dest Reg Number(DRN)。当 Valid 位为 1 时用来表示该 Entry 与那个通用寄存器建立联系。
- Value。该字段用来存放在重命名寄存器中的数值。
- Value Valid。该位为 0 时表示 Value 字段的数据无效(写指令没有执行完毕，或者该指令在 GIQ 或预约站中)，为 1 时表示 Value 字段中的数据有效(指令执行单元已将数据写入重命名寄存器中)。
- Latest bit 用来表示当前 Entry 中所代表的重命名寄存器是不是最新的。为 1 表示是最新的，为 0 表示不是最新的。有时同一个通用寄存器会被多次重命名。如下所示。

WAW-I1：　load r2，…，… ；

WAW-I2：　add r2，…，… ；

此时 r2 寄存器将使用两个重命名寄存器描述表的 Entry 对 r2 寄存器进行重新命名。

如表 2-9 所示，该重命名寄存器的第 7 个 Entry 表示，重命名寄存器 7 与通用寄存器 r10 相对应，该重命名寄存器中存放的值为 50。

3. 重命名寄存器的维护

处理器 C 在指令发射阶段将为每一条指令的目的操作数分配一个重命名寄存器，并为该指令分配一个 CRn 重命名寄存器，为简单起见，本书将不对 CRn 重命名寄存器进行说明。表 2-10 是在(R2 * R4)＋ R2 * (R2＋R2)程序执行时，处理器 C 使用重命名寄存器的例子。该表的上半部分是原始程序，从中可以发现指令 I2 存在 WAR 相关，指令 I3 存在 RAW 相关而指令 I4 存在 WAW 相关。

表 2-10　对(R2 * R4)＋ R2 * (R2＋R2)程序使用的寄存器进行重命名

指令顺序	指　令	指令描述	相关性
I1	Mult R10，R11，R2	R10 ← R11 * R2	
I2	Add R11，R3，R4	R11 ← R3 ＋ R4	R11 WAR

(续)

指令顺序	指　令	指令描述	相关性
I3	Mult R12, R11, R8	R12 ← R11 * R8	R11 RAW
I4	Add R12, R11, R9	R12 ← R11 + R9	R12 WAW

指令顺序	指　令	指令描述	相关性
I1	Mult R10′, R11, R2	R10′ ← R11 * R2	
I2	Add R11′, R3, R4	R11′ ← R3 + R4	R11 WAR
I3	Mult R12′, R11′, R8	R12′ ← R11′ * R8	R11 RAW
I4	Add R12″, R11′, R9	R12″ ← R11′ + R9	R12 WAW

处理器C在执行I1～I4这段指令时，需要首先对其进行寄存器重命名，如将R10替换为R10′，将R11替换为R11′，将R12替换为R12′与R12″，注意，R12寄存器在指令I3和I4中均作为目的操作数，因此重命名寄存器的描述表中有两个Entry描述R12寄存器。

当上述这段指令在处理器C中运行时，首先假定CQ队列为空，此时CQ0～3分别与I1～4对应，以保证程序按照顺序结束。

由于采用了寄存器重命名技术消除了WAR类型相关，指令I2′不需要等待指令I1′执行完毕即可运行，因此指令I1和I2可以同时在MU和SU1中运行。而指令I3和I4必须等待指令I2将寄存器R11′的结果计算完毕后，才能执行，因为重命名寄存器的技术无法解决RAW类相关。

在I3、I4指令中将不直接使用系统寄存器R11，而是重命名寄存器R11′，这种做法将会提高指令的执行效率，使用重命名寄存器R11′可以保证系统寄存器R11在没有被更新前就可以执行I3和I4，而不需要等待指令运行完毕再执行I3，I4指令。

指令I1将在指令执行完毕后释放重命名寄存器R10′，而R11′的释放需要注意同步，因为I3，I4仍然要使用R11′。此时可能的设计是在同步系统寄存器R11的时候，将重命名寄存器索引表中的Valid位清0。当I3和I4访问R11′时将检索重命名寄存器索引表，如果Valid为1则使用R11′，如果为0则使用R11。

在R11′寄存器的结果计算出后，I3和I4可以继续执行，此时需要将R12寄存器进行多次重命名，I3中的R12被命名为R12′，I4中的R12被命名为R12″。此时的结果如重命名寄存器的描述表所示，R12′使用重命名寄存器描述表的第9项(Latest bit为0)，而R12″使用重命名寄存器描述表的第13项(Latest bit为1)。淘汰Latest bit为1的重命名寄存器描述表项时需要保证Latest bit为0的表项淘汰后才能进行。

指令I3和I4因为采用了寄存器重命名技术消除了WAW相关，因此可以并行执行。在实际的IC设计时，重命名寄存器索引表和描述表较为复杂，这部分的具体实现也涉及一些专利技术，大多数厂商都没有将寄存器重命名的技术完全公开。这一部分与指令的流水线调度、多发射的实现密切相关，是指令运行的重要组成部分。

第3章 PowerPC处理器的内存体系结构

PowerPC处理器的内存体系结构由内存管理单元MMU、Cache的管理、系统总线的设计及其他一系列管理部件组成。内存体系结构是PowerPC处理器的设计核心。存储器访问的效率直接影响整个PowerPC处理器系统的效率,为此PowerPC处理器在设计其内存体系时,采用了一系列方法用来加速对存储器的访问,如猜测访问(Speculative Access)、对外设访问时保持Cache的一致性等方法。

本章将以PowerPC E500内核为例对PowerPC处理器的内存体系进行说明,其主要内容包括MMU的管理,L1 Cache的组成和E500内核使用的前端总线,即CCB(Core Complex Bus)总线。

3.1 PowerPC处理器的MMU

内存管理单元MMU是处理器的重要组成部分,主要功能是将程序使用的虚拟地址转换为处理器硬件能够直接访问的物理地址。在支持多进程的操作系统中,每个进程使用的虚拟空间独立,此时需要使用处理器提供的MMU将各个进程使用的地址空间进行有效隔离。

32位处理器能够访问的最大物理地址空间仅为4 GB,然而在有些操作系统中,如Linux系统,每一个进程都拥有3GB虚拟地址空间。因此在一个多进程操作系统中使用的虚拟地址空间一般都大于4 GB。为此操作系统必须采取某种转换机制建立虚拟地址空间与实际物理空间的映射。

在处理器的周围,除了具有可以直接进行访问的物理内存,如SRAM,SDRAM和DDR,还有许多外部设备,如PCI总线设备,IDE设备等。这些物理内存和外部设备都会占用处理器的物理地址空间。然而这些设备与处理器进行数据传送的方式往往与处理器访问内存的方式不同,如有些外部设备所需要的物理地址空间往往是不可Cache的。因此MMU必须合理控制和管理不同种类的物理地址空间的映射。

处理器的MMU为系统软件设立了转换查找表以支持虚实地址的映射。在这些查找表中将规定虚实地址的对应关系及访问控制等一系列信息。系统软件将利用处理器提供的查找表完成虚实地址的映射。

E500内核中包含内存管理单元MMU,E500内核MMU中共使用了两个查找表TLB0(Translation lookaside buffer)和TLB1来实现虚拟地址与物理地址的转换。其中TLB0用来进行页式映射,而TLB1用来实现段式映射。

E500内核分为V1和V2两个版本,这两个版本的MMU略有不同,在E500 V2内核中的物理地址为36位,所支持的最大物理地址空间为64 GB。而V1内核的物理地址为32位,所支持的最大物理地址空间为4 GB。此外V2内核的TLB0和TLB1的设置也与V1内核的TLB0和TLB1有些微小区别。

在PQIII系列处理器中,MPC8540/8560/8555/8541处理器使用E500的V1内核,而

MPC8548/8547/8545/8543 处理器使用 E500 的 V2 内核。

本节将主要介绍 V1 内核的 MMU。同时将 E500 内核的 MMU 与 PowerPC 603E 的 MMU 进行简单比较。

3.1.1 E500 V1 内核的虚实地址转换

E500 V1 内核是一个 32 位处理器的内核,其最大物理地址空间为 4 GB。下文将 E500 V1 内核简称为 E500 内核。E500 内核在内部可以使用 41 位的临时虚拟地址(Interim Virtual Address),因此 E500 内核所能直接支持的最大虚拟地址空间为 1 TB。E500 内核 MMU 的主要作用就是将 1TB 的虚拟地址空间映射到 4 GB 的物理地址空间中。E500 内核的 41 位虚拟地址组成如图 3-1 所示。

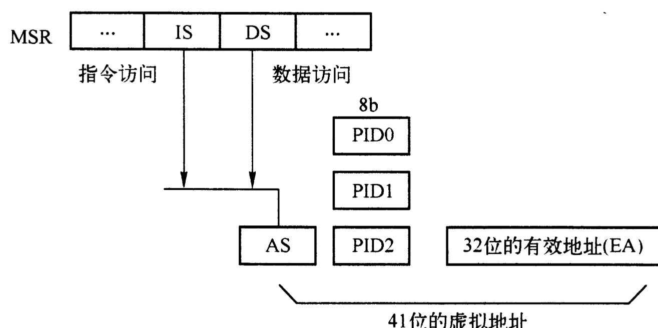


图 3-1 E500 内核的虚拟地址组成

从图 3-1 可知,E500 内核的 41 位虚拟地址由 1 位 AS,8 位 PID 和 32 位的 EA 组成。其中各个位的描述如下。

(1) AS 位。该位来自 E500 内核 MSR 寄存器的 IS 位或者 DS 位。当 E500 内核访问指令空间时 AS 位为 MSR 寄存器中的 IS 位,当访问数据空间时 AS 位为 MSR 寄存器 DS 位。

在 E500 内核中根据 AS 位的不同将地址空间分为两种,分别为地址空间 0 和地址空间 1。当 AS 位为 0 时,E500 内核使用地址空间 0,当 AS 位为 1 时,E500 内核使用地址空间 1。当 E500 内核进入中断或者其他异常处理程序时,MSR 寄存器的 IS 位和 DS 位将被清零,因此中断及其他异常的处理程序必须运行在 E500 的程序地址空间 0 中,在这段程序中所访问的数据也必须要在数据地址空间 0 中。在 E500 内核中设置两个地址空间的本意是让系统软件使用地址空间 0,而应用软件使用地址空间 1。

然而,Linux PowerPC 没有使用地址空间 1,所有指令、数据访问都将在地址空间 0 中,即虚拟地址的 AS 位在 Linux 系统运行过程中始终为 0。Vxworks 系统使用了 E500 内核提供的地址空间 0 与地址空间 1。

(2) PID 字段。E500 内核一共支持 3 个 8 位的 PID 寄存器用来保存当前进程的 ID。PID 寄存器的使用可以对不同的进程空间进行保护。目前 Linux PowerPC 并不支持 E500 内核的 PID 寄存器,而简单地将 TLB 中所有 Entry 的 TID 位置为 0 以忽略对 PID 寄存器的检查。实际上在 Linux PowerPC 中,可以使用 PID 字段保存每一个进程的 PGD 指针,从而加速进程的切换速度,但是 Linux PowerPC 为统一起见并没有使用 PID 字段保存 PGD 指针,而是将每一个进程的 PGD 指针保存在物理内存中。Linux PowerPC 并没有充分利用 E500 内核提供的

MMU。

(3) EA(Effective Address)字段。该字段用来存放被访问数据的有效地址。E500 内核的有效地址为 32 位。有效地址 EA 是在程序中能够直接使用的地址。E500 内核的 EA 由有效地址页表索引 EPN(Effective Page Number)和地址偏移 Offset 两部分组成。

比如在指令“lwz r4, 8(r5)”中 r5 寄存器中存放的地址就是有效地址。E500 内核的有效地址为 32 位。在 Linux PowerPC 中,E500 内核虚拟地址的 AS 位和 PID 字段并没有被使用,因此数据的有效地址等效于该数据的虚拟地址。

E500 内核中使用了两级 MMU 结构,L1-MMU 和 L2-MMU,以及一些辅助寄存器和指令用以支持虚实地址转换。E500 内核的内存管理与其他 PowerPC 体系的内存管理有较大区别,在 E500 内核中地址转换不能被禁止,并在内部支持两个地址空间,分别为地址空间 0 与地址空间 1,E500 内核的虚实地址转换流程如图 3-2 所示。

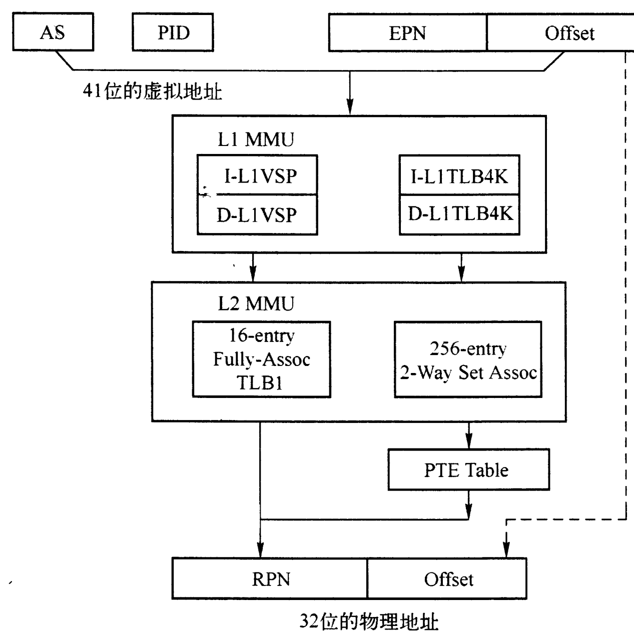


图 3-2 E500 内核的虚拟地址组成

603E 内核的 MMU 可以被关闭,当处理器进入异常和中断处理程序时,MMU 将自动被关闭,从而处理器可以对物理地址空间直接访问。

E500 内核的 32 位物理地址由物理地址页表索引 RPN(Real Address page Number)和偏移组成。E500 内核在使用 MMU 进行虚实地址转换时,只需要获得物理地址的 RPN 参数,而物理地址的偏移 Offset 参数将从虚拟地址中直接获得。

L1 MMU 包括 I-L1VSP,D-L1VSP,I-L1TLB4K 与 D-L1TLB4K。I-L1VSP,D-L1VSP 含有 4 个 Entry,采用全相联的结构。I-L1TLB4K,D-L1TLB4K 含有 64 个 Entry,采用 4 路组相联的结构。在 L1-MMU 中,I-L1VSP 和 D-L1VSP 是 TLB1 的数据缓冲,而 I-L1TLB4K 和 D-L1TLB4K 是 TLB0 的数据缓冲。

E500 内核中设置 L1 MMU 有以下两个作用:

- L1 MMU 的访问速度较快。在多数情况下 E500 内核进行虚实地址转换时，虚拟地址都可以在 L1 MMU 中命中，从而提高虚实地址转换的效率。
- E500 内核的 L1 MMU 采用程序空间的地址转换与数据空间的地址转换分离的方法。L1 MMU 的这种哈佛结构可以提高指令空间与数据空间进行地址转换的效率。在 PowerPC 603E 内核中没有采用 L1 MMU，而是直接使用 IBAT 和 DBAT 实现块式内存映射，将指令空间与数据空间的虚实地址转换分离，并采用 SR 寄存器和 TLB 实现页式映射。其中 IBAT 和 DBAT 的作用与 E500 内核的 TLB1 类似，而 SR 寄存器和 TLB 的作用与 TLB0 类似。

E500 内核的 L2 MMU 包括 TLB0 和 TLB1。其中 TLB1 含有 16 个 Entry，而 TLB0 含有 256 个 Entry。进行地址虚实转换时，如果虚拟地址在 L1 和 L2 MMU 中都没有命中，将引发 TLB Miss 异常，并在异常程序中使用 PTE 表继续进行虚实地址转换，PTE 表存放在物理内存中，不同的操作系统对 PTE 表的处理不尽相同。

在 E500 内核中，一个完整的虚实地址转换步骤如下所示：

(1) E500 内核在进行地址转换时将首先确定是对程序地址还是数据地址进行转换。如果是进行程序的地址转换，E500 内核将首先查找 I-L1VSP 和 I-L1TLB4K，否则将查找 D-L1VSP 和 D-L1TLB4K。

(2) 在 L1 MMU 中根据当前虚拟地址，查找物理地址的 RPN。如果当前虚拟地址在 L1 MMU 中命中，则获得对应的 RPN，并结束虚实地址转换。如果当前虚拟地址没有在 L1 MMU 中命中，则继续在 L2 MMU 中查找。

(3) 在 L2 MMU 的 TLB0 和 TLB1 中继续根据虚拟地址，查找物理地址的 RPN。在一个基于 E500 内核的处理器系统中，会根据实际需要将 TLB1 进行初始化，TLB1 在完成最终的初始化后，一般不会变化，而 TLB0 将会被动态调整。在进行虚实地址转换时，不能出现两个虚拟地址同时出现在 TLB0 或者 TLB1 中，如果出现这种现象可能会导致不可预料的错误。如果当前虚拟地址在 TLB0 或者 TLB1 中命中，E500 内核获得相应的 RPN，并结束虚实地址转换。

(4) 如果当前的虚拟地址在 TLB0 或者 TLB1 中都没有命中，E500 内核将会产生 ITLB Miss 或者 DTLB Miss 异常，之后异常处理程序将搜索在物理内存中的 PTE 表，获得相应的 RPN 后对 TLB0 进行更新，同时获得 RPN。

(5) 如果在 PTE 表中还没有获得合适的 RPN，将进一步对虚拟地址进行分析，这一过程较为复杂。Linux 系统使用 page _ fault 函数处理这种情况，该函数将在下文详细介绍。

3.1.2 L1 MMU 和 L2 MMU 中的 Entry

E500 内核的 L1 MMU 和 L2 MMU 中包含了许多 Entry，在这些 Entry 中又包含了许多字段用来支持虚实地址的转换。系统软件不能直接操作 L1 MMU 的 Entry，而是通过操作 L2 MMU 的 Entry 来影响 L1 MMU 的 Entry。

L1 MMU 和 L2 MMU 的 Entry 包含以下字段和位。

(1) V 位。V 位用来表示当前 Entry 是否有效，当 V 位为 0 时，表示当前 Entry 无效；为 1 时，表示当前 Entry 有效。在系统复位完成时，L1 MMU 和 L2 MMU 中所有 Entry 的 V 位为 0，表示所有的 Entry 都无效。

(2) TS位。TS位为0时,表示当前的Entry描述的地址空间属于有效地址空间0;为1时,表示当前的Entry描述的地址空间属于有效地址空间1。

(3) TID[0:7]字段。E500内核进行虚实地址转换时,会将TID字段与寄存器PID[0~2]中存放的数据进行比较,如果存放在PID寄存器的值与TID字段都不相等,则产生TLB Miss异常。当TID字段为0时,E500内核将忽略PID寄存器与TID字段的比较。Linux PowerPC将TID字段设置为0。

(4) EPN[0:19]字段。该字段用来存放当前Entry描述的有效地址页表号。对于TLB1,该字段随SIZE字段的不同,其有效位数不同。对于TLB0,该字段的所有位有效。

(5) RPN[0:19]字段。该字段用来存放当前Entry描述的物理地址页表号。对于TLB1,该字段随SIZE字段的不同,其有效位数不同。对于TLB0,该字段的所有位有效。

(6) SIZE[0:3]字段。SIZE字段用来存放当前Entry的页表大小。E500内核的TLB0只支持固定大小的页表尺寸4 KB,其SIZE字段的值只能为0001;而TLB1支持的页表尺寸可变,SIZE字段的值可以为0001~1001。

(7) PERMIS[0-5]字段。该字段用来描述当前Entry的访问控制位,共由6位组成。分别是UX位(User-mode Execution),该位为1时表示此Entry描述的空间在E500内核处于用户模式时可以执行程序;SX位(Supervisor-mode Execution),该位为1时表示在E500内核处于超级用户模式时可以执行程序;UW(User-mode Write)位,UR(User-mode Read)位,SW(Supervisor-mode Write)位和SR(supervisor-mode read)位,此四位为1时分别表示在E500内核处于用户模式和超级用户模式时可写、可读。

(8) WIMGE字段。该字段是Entry中的重要字段。其中E位用来确定当前Entry描述的内存空间存放数据是使用大端还是小端方式,E500内核可以为每一个页面区间进行端模式选择。而许多PowerPC处理器,如603E内核只能设置MSR寄存器中的LE位用来选择整个处理器中所用的内存空间是采用大端模式还是小端模式,而不能分别对每一段地址空间进行设置。E500内核的MSR寄存器中没有LE位。这里提醒读者注意,PowerPC处理器基本上都使用大端模式。WIMG四位将在下文中详细介绍。

(9) X0,X1位。这两位用来描述当前Entry的一些额外属性,E500内核没有规定系统软件将如何使用这两位。

(10) U[0:3]字段。系统软件可以使用该字段用于一些自定义的用途。

(11) IPROT位。当IPROT位为1时,表示当前Entry被保护,系统软件不能使用tlbivax指令或者对MMUCSR0寄存器进行操作,将该Entry使无效。该位只能被tlbwe指令清零或者置1。E500内核中,只有TLB1支持该位。在TLB0的Entry中,IPROT位只能为0。

E500内核中的虚拟地址使用L1 MMU,TLB0或TLB1中的Entry进行地址转换时,要根据以下条件,判断当前虚拟地址是否在TLB的Entry中命中。

- 首先参与比较的Entry的V位必须有效。
- Entry中的TS位与MSR的寄存器的IS或DS位相等。
- Entry的TID字段需要与PID0,PID1或者PID2寄存器中任意寄存器的内容相等。当TID字段为0时,忽略与PID寄存器的比较。
- Entry中的EPN字段是否与虚拟地址的EPN相等。
- Entry中的PERMIS字段满足当前对虚拟地址的操作。

当一个虚拟地址满足上述所有条件时，该虚拟地址在 TLB0 或者 TLB1 中的某个 Entry 中命中，此时可以获得该 Entry 的 RPN 字段。将此 RPN 字段与虚拟地址的 Offset 字段级联就可以获得该虚拟地址的物理地址，从而完成虚拟地址到物理地址的转换。

3.1.3 与 MMU 管理相关的寄存器

E500 内核中设置了一系列寄存器用来对 MMU 进行维护与更新，这些寄存器包括对 MMU 进行管理和配置的寄存器和对 TLB 中的 Entry 进行配置和管理的寄存器。

1. PID0～2 寄存器

PID0～2 寄存器(Process ID registers)用来存放当前进程有效地址的进程 ID 号，该寄存器的第 54～63 字段共 8 位有效。由 3.1.1 节可知，E500 内核的 41 位虚拟地址由 AS，PID 字段和有效地址 EA 组成。该组寄存器就是用来存放虚拟地址的 PID 字段。

在 Book E 中只有 PID0 寄存器有效，PID1～2 寄存器只在 E500 内核中有效。

2. MMUCSR0 寄存器

MMUCSR0(MMU Control and Status Register 0)寄存器用来使无效 TLB0 和 TLB1 的所有 Entry，即将 Entry 中的 V 位清零。该寄存器一共由两位组成。

- L2TLB0 _ FI 位。对此位写 1 时将使无效 TLB0 内的所有 Entry。
- L2TLB1 _ FI 位。对此位写 1 时将使无效 TLB1 内的所有 Entry。

当对以上各位写 1 时将使无效 TLB 内的所有 Entry，当使无效操作结束后，该位将会被自动清零。

3. MMUCFG 寄存器

MMUCFG(MMU Configuration Register)寄存器用来保存当前 MMU 管理单元的配置信息，包括 PID 寄存器的数量，大小，TLB 的数量及 MMU 体系结构的版本号，此寄存器只读。

4. TLB0CFG 和 TLB1CFG 寄存器

其中 TLB0CFG 和 TLB1CFG 寄存器用来描述 TLB0 和 TLB1 的结构信息包括 TLB 的 Entry 数量，页表的尺寸等信息。这两个寄存器只读。

5. MAS0～4 及 MAS6～7 寄存器

MAS0～4 及 MAS6～7(MMU Assist Registers)寄存器的主要作用是维护 MMU 管理单元中 TLB 的各个 Entry。

寄存器 MAS1～3 中存放的各个字段与 TLB 中 Entry 的各个字段一一对应。系统软件在对 TLB 的 Entry 进行写操作时，将根据 MAS0 寄存器的值将存放在 MAS1～3 寄存器中的字段写入到相应 Entry 的各个字段中。E500 内核使用指令 tlbre 和 tlbwe 对 TLB 中的 Entry 进行读写操作。

寄存器 MAS0 的主要字段的含义如下所示。

(1) TLBSEL 字段。该字段用来选择即将操作的 TLB，为 0 时表示使用 TLB0；为 1 时表示使用 TLB1。

(2) ESEL 字段。ESEL 字段用来选择 TLB 中的 Entry。当使用 TLB1 时，ESEL 字段中的低 4 位有效，用来选择 TLB1 的 Entry 数，TLB1 只有 16 个 Entry。

当使用 TLB0 时，ESEL 字段只有最低一位有效，用来选择 TLB0 的组，E500 内核的 TLB0 使用 2 路组相联结构，此时 E500 内核还需要使用 MAS2 寄存器的 EPN 字段的第 45～51 位选

择在不同组内的Entry。E500 V1内核中，TLB0使用了256个Entry，因此一共需要8位来表示TLB0的Entry数。

(3) NV位。NV位对于TLB1没有意义。E500内核中，只有NV字段的最低位有效。在执行TLB写操作时NV位将被写入TLB0的相应Entry中。在TLB0的所有Entry都被使用而且发生了TLB Miss后，需要替换TLB0中的Entry，此时E500内核使用此位确定如何对TLB0中的Entry进行替换。

寄存器MAS1～3的各个字段与TLB中Entry中的各个字段一一对应，且含义相同，包括V位，IPROT位，TID字段，TS位，TSIZE字段，EPN字段，X0位，X1位，WIMEG字段，RPN字段，U[0:3]和PERMIS字段。

寄存器MAS4用来存放当TLB miss异常发生时，对MAS0～3寄存器自动加载而使用的默认值。E500内核为提高TLB Miss异常的处理效率，当TLB Miss异常出现时，E500内核将自动把MAS0～2寄存器的值使用寄存器MAS4中的值替换，如下所示。

- MAS0[TLBSEL] ← MAS4[TLBSELD]
- MAS1[TID] ← MAS4[TIDSELD]
- MAS1[TSIZE] ← MAS4[TSIZED]
- MAS2[X0,X1] ← MAS4[X0D,X1D]
- MAS2[WIMGE] ← MAS4[WD,ID,MD,GD,ED]

一般来说MAS4中的TLBSELD为0，表示使用TLB0处理TLB miss异常。TSIZED为4KB，与TLB0中支持的页表大小相等。

MAS6寄存器用于对TLB表进行检索，E500内核使用tlbsx指令对TLB表进行搜索。tlbsx指令的格式为“tlbsx RA, RB”。

tlbsx指令对TLB中的Entry进行检索时，将使用MAS6寄存器中的SPID0字段和SAS位与tlbsx指令中的RA+RB(或者RB)组成的有效地址级联，然后组成一个41位的虚拟地址后再对TLB进行检索。

3.1.4 与MMU管理相关的指令

E500内核中设置了一系列指令用来对MMU进行维护与更新，这些指令包括对TLB中的Entry进行配置和管理的指令，如tlbre，tlbwe，tlbsx，tlbivax和tlbsync等。

1. tlbre指令

tlbre指令用于读取TLB中的相应Entry。tlbre指令一般用作系统的维护诊断，在Linux PowerPC中tlbre指令仅在系统初始化中得到了使用。在使用tlbre指令读取TLB的Entry前需要对MAS0进行设置。

tlbre指令不使用操作数，而是使用MAS类寄存器与TLB进行数据交换。

- 输入：MAS0寄存器的TLBSEL和ESEL字段及MAS2寄存器的EPN字段。

注意EPN字段在对TLB0的Entry进行读时有意义，而对TLB1的Entry进行读操作时EPN字段将被忽略，因为TLB1只有16个Entry，使用4位的ESEL字段就足以确定相应的Entry。

对TLB0的Entry进行操作时，tlbre指令将根据MAS0寄存器的ESEL和MAS2寄存器的EPN字段确定TLB0的Entry号，并使用此Entry号进行TLB相应Entry读操作。

● 输出:将指令 TLB 的相应 Entry 的各个字段放入 MAS1,MAS2 和 MAS3 寄存器中。

2. tlbwe 指令

tlbwe 指令用于设置 TLB 中相应的 Entry。

● 输入:MAS0 寄存器的 TLBSEL, ESEL 和 MAS1～3 寄存器中与 TLB Entry 对应的字段。

● 输出:无。

3. tlbsx 指令

tlbsz 指令的格式为"tlbsx RA, RB"。tlbsx 指令用来在 TLB 中查找与指定有效地址对应的 Entry,并将相应 Entry 的内容放入 MAS 寄存器中。

tlbsx 指令使用 RA + RB(或者 RB)产生的有效地址和 MAS4 寄存器中存放的字段,对 TLB 表进行查找,如果命中,则将相应 Entry 的内容放入 MAS0～3 寄存器中。E500 内核要求 RA 寄存器存放的数据为 0。

4. tlbivax 指令

tlbivax 指令的格式为"tlbivax RA, RB"。tlbivax 指令的主要作用是使用 RA + RB(或者 RB)产生的有效地址和 MAS4 寄存器中存放的字段,对 TLB 表进行查找,然后对 TLB 相应的 Entry 进行使无效操作。如果 Entry 的 IPROT 位为 1 时 tlbivax 指令将不能对将此 Entry 的状态置为无效。

5. tlbsync 指令

tlbsync 指令用于同步对 TLB Entry 的读写。该指令将会引发系统总线的 TLBSYNC 周期。其主要用途是将 tlbivax 指令的更新 Entry 信息广播到系统总线上,以同步在系统总线上的其他处理器的内存系统。该指令仅在 SMP 处理器中有效。

提醒读者注意,只有 E500 内核的 HID 寄存器中 ABE 位被置为 1 后,tlbsync 指令才能在系统总线上进行广播操作。

3.1.5 E500 内核的 TLB1

在 E500 内核中,TLB1 用来实现虚拟地址与物理地址之间的段映射方式。在 E500 内核复位后,TLB1 中除了 Entry0 之外,其他的 Entry 都被置为无效。TLB1 的 Entry 0 初始值如表 3-1 所示。

表 3-1 E500 内核初始化后 TLB1 的 Entry 0

字 段	复 位 值	注 释
V	1	表示当前 Entry 有效
TS	0	使用地址空间 0
TID[0:7]	0x00	不与 PID 寄存器进行比较
RPN	0xFFFFF	将有效地址空间 0xFFFFF000～0xFFFFFFFF 与物理地址空间 0xFFFFF000～0xFFFFFFFF 相映射,大小为 4 KB
EPN	0xFFFFF	
SIZE[0:3]	0001	
SX/SR/SW	111	处理器运行在超级模式时可以对此空间读写执行
UX/UR/UW	000	处理器运行在用户模式时不可以对此空间读写执行

（续）

字　段	复 位 值	注　释
WIMGE	01000	对此空间的操作不使用 Cache，不执行存储器一致性操作，不对存储器进行保护，并采用大端模式
X0-X1	00	
U0-U3	0000	
IPROT	1	此 Entry 受保护，不会被 tlbivx 指令使无效

E500 内核的系统复位地址是 0xFFFFFFFC。此时系统只能访问从 0xFFFFF000 开始的 4KB 地址空间。在这段程序空间里，引导程序需要完成对 TLB1 的其他 Entry 初始化操作，建立基本的地址映射。

TLB1 支持可变长度页表，可支持的页表大小如表 3-2 所示，TLB1 中一共有 16 个 Entry，支持的最大内存地址空间为 4 GB。改变 TLB1 相应 Entry 的 SIZE 字段将会调整 TLB1 中所支持的页表大小。

当 TLB1 的页表大小发生变化时，进行虚实地址转换需要进行比较的 EPN 字段的有效位数，Offset 字段的有效位数也随之改变。

表 3-2　TLB1 的 EPN 与 Size 和 Offset 字段之间的关系

EPN 的有效位数	Offset 的有效位数	Size	Page Size
—	—	0000	Reserved
20	12	0001	4 KB
18	14	0010	16 KB
16	16	0011	64 KB
14	18	0100	256 KB
12	20	0101	1 MB
10	22	0110	4 MB
8	24	0111	16 MB
6	26	1000	64 MB
4	28	1001	256 MB
E500 V1 内核不支持		其他	—

系统软件可以使用 TLB1 映射 I/O 空间，如 PCI 总线地址空间，RapidIO 地址空间，局部地址总线空间。也可以使用 TLB1 对主存储器，如 SDRAM、DDR，进行地址映射。使用 TLB1 进行映射的优点是配置简单，一次可以将最多 256 MB 的空间进行映射。对于许多简单的应用而言，系统软件设计可以只使用 TLB1 完成虚实地址空间的转换。而不需要使用 TLB0 进行页式虚实地址的转换。

考虑一个基于 MPC8540 处理器的嵌入式系统，在此系统中使用了 512 MB 的 DDR 作为主存储器，16 MB 的 Flash 作为程序空间，需要支持 256 MB 的 PCI 空间，256 MB 的 RapidIO 空间，32 MB 的外部设备空间与 1 MB 大小的 MPC8540 内部存储器映射的寄存器空间。其物

理地址空间如表 3-3 所示。

对于表 3-3 中的处理器系统，系统软件可以方便地使用 TLB1 对以上空间进行映射。首先系统软件将 MSR 中的 IS 和 DS 位清零，只使用 E500 内核的地址空间 0，同时将 TLB1 Entry 中的 TS 字段清零。此时 E500 内核的有效地址将等效为虚拟地址。之后使用 E500 内核中的 TLB1 中的 6 个 Entry 对这些存储器空间进行映射，如表 3-4 所示。

表 3-3　一个处理器系统的存储器映像

名　称	大小/MB	物理地址范围
DDR	512	0000-0000～1FFF-FFFF
RapidIO	256	4000-0000～4FFF-FFFF
PCI	256	6000-0000～6FFF-FFFF
其他外设	32	A000-0000～A1FF-FFFF
内部寄存器空间	1	E000-0000～E00F-FFFF
Flash	16	FF00-0000～FFFF-FFFF

表 3-4　使用 TLB1 进行虚实地址转换

TLB1 Entry	名　称	EPN	RPN	大小/MB
0	DDR1	0x00000	0x00000	256
1	DDR2	0x10000	0x10000	256
2	RapidIO	0x40000	0x40000	256
3	PCI	0x60000	0x60000	256
3	其他外设	0xA0000	0xA0000	64
4	内部寄存器空间	0xE0000	0xE0000	1
5	Flash	0xFF000	0xFF000	16

采用表 3-4 对 TLB1 进行配置，可以将 E500 内核的有效地址与物理地址一一对应，此时 MPC8540 中的数据和地址的有效地址与物理地址的值相等。值得注意的是一个 TLB1 的 Entry 最多只能映射 256 MB 的地址空间，因此至少要使用两个 Entry 才能对 512 MB 的 DDR 空间进行映射。由于 TLB1 的 Entry 不支持 32 MB 大小的页表，因此对 32 MB 的其他外设空间进行映射时，或者使用 2 个 16 MB 大小的 Entry 或者使用一个 64 MB 大小的 Entry 进行映射。注意在 TLB1 中进行映射的虚拟地址不能重叠。

对于简单的处理器系统，系统设计者可以只使用 TLB1 进行空间映射，只采用 TLB1 进行虚拟地址到物理地址的转换有许多优点，如使用 TLB1 进行虚实地址转换时，不会出现 TLB Miss 的情况，能够在一定程度上提高系统的效率。但是对于较为复杂的系统仅使用 TLB1 无法完成所有虚拟地址空间的映射，原因如下：

- TLB1 最多只能对 4 GB 大小的虚存空间进行映射，当有些操作系统使用的虚拟地址空间大于 4 GB 时，就不能够只使用 TLB1 进行映射。
- TLB1 中最多只有 16 个 Entry。但是在某些复杂的应用中，内存空间的种类可能大于 16 种，此时仅使用 TLB1 不能满足系统的要求。

在以上几种情况下，系统软件必须采用其他方式进行虚拟地址空间到物理地址空间的转换。还有一种情况，需要读者考虑，如果你使用的系统，其地址空间能够完全使用 TLB1 进行

映射，那么此时 TLB1 仅提供了一种一一对应的地址转换方式而已。此时系统 MMU 的地址转换功能并没有起到太大的作用，在这种情况下实际上关闭了 MMU 也没有什么关系。只是在 E500 内核中，MMU 中的地址转换单元不能被关闭，系统软件必须对 TLB1 中的 Entry 进行配置。

3.1.6 E500 内核的 TLB0

E500 内核中，TLB0 是 MMU 地址转换单元的核心。E500 内核的 TLB0 共支持 256 个 entry，采用 2 路组相连结构，所支持的页表大小为 4 KB。因此 TLB0 所能直接覆盖的物理地址空间只有 1 MB，但是系统软件只是将 TLB0 作为系统页表的缓冲。

采用 TLB0 进行虚存管理的系统需要在主存储器中建立页表，页表中的每一个 Entry 所能管理的页表大小也为 4 KB，与 TLB0 支持的页表大小相等。

在一个复杂的操作系统中，一般会用 TLB0 实现虚拟地址到物理地址的转换。如 Linux PowerPC 中也使用了 TLB0 进行虚实地址的转换。在 Linux 系统中，Linux 内核将占用 1 GB 空间，而每一个用户进程都将具有 3 GB 空间，此时对用户进程地址空间的管理必须使用 TLB0 空间。

由于在 TLB0 中能直接映射的地址空间只有 1 MB，当一个程序使用的地址不在这 1 MB 空间时，将产生 TLB Miss 异常。E500 内核中一共有两种 TLB Miss 异常，一种是指令 TLB Miss 异常，一种是数据 TLB Miss 异常。

E500 内核中，TLB0 中直接映射的地址空间只有 1 MB，但是由于程序执行的局部性，进程使用的程序空间和数据空间在 TLB0 中命中的概率非常高。

E500 内核为 TLB0 设置了 256 个 Entry。但是用户不必因为 TLB0 中只有 256 个 Entry，而担心使用 TLB0 进行虚实地址的转换效率。因为 E500 内核只设置 256 个 Entry 是建立在许多理论和实验数据基础之上的，是建立在认真的量化分析的基础之上的。

虽然使用 TLB0 进行虚实地址转换的效率较高，但是在进程运行中还是会产生一部分 TLB Miss 异常。E500 内核为提高 TLB Miss 异常的处理效率，当 TLB Miss 异常出现时，会自动将 MAS0～2 寄存器的值使用寄存器 MAS4 中的值替换（参见本章 3.1.3 节），然后进入 TLB Miss 异常处理程序。

TLB Miss 异常处理程序将对存放在外部存储器中的页表进行检索，然后查找到页表中合适的 Entry，将此 Entry 中的值更新到即将被替换的 TLB0 的 Entry 中。在 E500 内核中 TLB0 使用 2 路组相联的方式进行管理，因此有两个可能的 Entry 可以被替换。在 E500 内核中使用 LRU 算法进行 TLB0 的 Entry 替换，如图 3-3 所示。

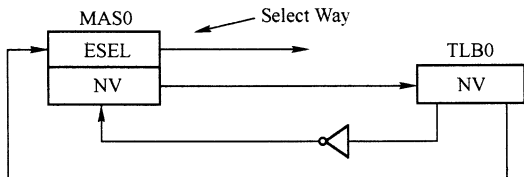


图 3-3 TLB0 的 Entry 替换机制

E500 内核的 TLB0 中包含一个 NV 位。该位用来实现 TLB0 的 Entry 替换。如图 3-3 所

示，TLB0 进行替换时将当前 TLB0 中的 NV 位及其反码($\overline{NV}$)分别写入 MAS0 寄存器中的 ESEL 字段和 NV 位中，然后将 MAS0 寄存器中的 NV 位赋值到 TLB0 的 NV 位中。因为 E500 内核的 TLB0 中可能被替换的 Entry 只有两个，因此这种算法也等效于 LRU 算法。E500 V2 内核进行 TLB0 的替换时，也使用 LRU 算法，但是其替换流程远比图 3-3 所示的算法复杂，因为 E500 V2 内核使用 4 路组相联对 TLB0 中的 Entry 进行管理。

在进入 TLB Miss 异常处理程序后，MAS1～3，MAS6 寄存器中还有些字段将被自动加载，如下所示：

- MAS1 寄存器中的 V 位将置 1，IPROT 字段将清零，TS 位将使用 MSR 寄存器中 IS 或 DS 的值，如果是指令 TLB miss 使用 MSR 寄存器中 IS 位的值，否则使用 MSR 寄存器中的 DS 位。
- MAS2 寄存器中的 EPN 将使用引发 TLB miss 异常的有效地址的 EPN。
- 将 MAS3 寄存器中的 RPN，PERMIS，U0～U3 字段清零。
- MAS6 中的 SPID0 字段将使用处理器的 PID0 寄存器的值，SAS 位使用 MSR 寄存器中 IS 或 DS 的值，如果是指令 TLB miss，该位使用 MSR 寄存器中 IS 位的值，否则使用 MSR 寄存器中的 DS 位。

由于 E500 内核在处理 TLB miss 时已提前将 MAS 寄存器中的许多字段准备好，系统软件只需要从系统页表中获得 RPN，PERMIS，U0～U3 字段并将其存放在 MAS3 寄存器中，然后使用 tlbwe 指令更新 TLB0 的相应 Entry，即可以完成对 TLB0 中相应 Entry 的更新工作。

在完成对相应 Entry 的更新后，系统程序将很快结束 TLB Miss 异常处理函数。在异常处理程序完成对 TLB0 的 Entry 的更新后，E500 内核将从该 Entry 中获得数据或程序所需的物理地址。

TLB0 为系统软件管理内存提供了一种灵活的方式，但是由于 TLB0 miss 的存在，系统软件有时必须检索存放在外部存储器的页表才能获得所需物理地址。因此在某种程度上说，使用 TLB0 进行地址映射会降低系统的效率。但是系统软件的设计有时必须在灵活性和效率中进行取舍。在 Linux 系统中需要采用 TLB1 和 TLB0 相结合的方法进行地址映射。

在基于 E500 内核的操作系统中，使用 TLB1 进行虚实地址转换是必不可少的，比如在使用 TLB0 进行虚实地址映射时，将使用存放在外部存储器的页表。当系统软件访问此页表时，这个页表必须使用 TLB1 进行虚实地址转换，否则将可能出现在 TLB Miss 异常处理程序中访问再次出现数据 TLB Miss 的情况，从而导致整个系统的崩溃。

在 Linux PowerPC 中，使用了 TLB1 和 TLB0 相结合的方法实现虚实地址转换。

3.2 E500 内核的 Cache 的组成

Cache 的主要作用是作为外部存储器的缓冲，用以减缓存储器访问的瓶颈问题，从而提高整个处理器的使用效率。

高速缓冲 Cache 位于处理器和主存储器之间，它的容量比内存小，但存取速度高，其内容为主存储器的部分拷贝。在程序运行过程中，当需要取指令或取数时，处理器首先检查缓存中是否有此内容，若有就从缓存中取出，若没有再从存储器取出。

在处理器中可能存在一级、二极或者三级高速缓存(L1，L2 或者 L3 Cache)，个别系统还

有四级缓存(L4 Cache)。IBM 的 xSeries 服务器使用的处理器使用了 L4 Cache。

L1 Cache 和 L2 Cache 是一个处理器重要的组成部分。处理器中内置的 L1 Cache 可以有效提高处理器的运行效率。L1 Cache 的容量和结构对处理器的性能影响较大。

而 L2 Cache 的作用是为了协调处理器运行速度与内存存取速度之间的差异,L2 Cache 对提高处理器的运行性能也有很大的帮助。L2 Cache 实际上是处理器和主存储器之间的真正缓冲。L2 Cache 的大小一般为 128 KB,256 KB 或 512 KB。L2 Cache 的大小对于处理器运算性能并没有太大的帮助,但对于一些实际应用程序,L2 Cache 容量的提高可以直接带来整体性能的改变。

E500 内核中仅包含 L1 Cache,而不包含 L2 Cache。目前基于 E500 内核的处理器,如 MPC8541,MPC8548,包含了 L2 Cache。但是这些 L2 Cache 并不属于 E500 内核,而是属于处理器。

在现代处理器系统中对 Cache 采取了许多不同的处理方法。例如虚拟 Cache 技术直接使用虚拟地址进行 Cache 访问,然后在 Cache 中做出相应的虚实地址的转换和管理工作。这种技术的实现较为复杂,尤其是在多进程进行地址空间切换时,虚拟 Cache 的同步处理较为复杂,不利于系统总线和处理器频率的提高,因此目前绝大多数的处理器内核没有使用这种虚拟 Cache 技术。

E500 内核也没有使用虚拟 Cache 技术。因此 E500 内核对 Cache 的访问所使用的地址为物理地址,相对虚拟 Cache 技术,这种实现要简单得多。当处理器对 Cache 进行访问时首先要通过 MMU 管理单元获得指令或数据的物理地址,然后通过 L1 Cache 以及处理器内核的整个存储系统对最终数据进行访问。

3.2.1 L1 Cache 的结构

E500 内核的 L1 Cache 采用哈佛结构将指令 L1 Cache 与数据 L1 Cache 分离。指令与数据 L1 Cache 的大小都为 32 KB,并采用 8 路组相联结构,Cache 行长度为 256 位,L1 Cache 的组数为 128。对于一个采用多路组相联结构的 Cache,Cache 的大小、Cache 行长度、路数与组数之间的关系如以下公式所示:

Cache 大小(32 KB) = Cache 行长度 (32B) × 路数 (8 路) × 组数 (128 组)

E500 内核在每一个 Cache 行中包含一个地址标签(Address Tag),四个状态位和 32 B 的数据,如图 3-4 所示为数据 L1 Cache 的组织结构。指令 L1 Cache 的组织结构与数据 L1 Cache 的组织结构类似。

E500 内核的 L1 Cache 使用 15 位地址对 32 KB 大小的数据 L1 Cache 空间进行编码。E500 内核 L1 Cache 使用 8 路组相联结构,因此使用 3 位表示路数,使用 7 位表示组数。由于 Cache 行长度为 32 B,所以还要使用 5 位对 Cache 行中的数据进行选择。如下所示:

第 0~2 位	第 3~9 位	第 10~14 位
Way Number(路数)	Set Number(组数)	Cache Line

如果一个处理器使用 4 路或 2 路组相连结构,而且 Cache 的大小依然为 32 KB 时,Way Number 字段降低为 2 位或者 1 位,并将 Set Number 字段扩大 2 位或者 1 位;如果使用 16 路或 32 路组相连结构时需要将 Way number 字段扩大为 4 位或 5 位,并将相应的 Set Number 字

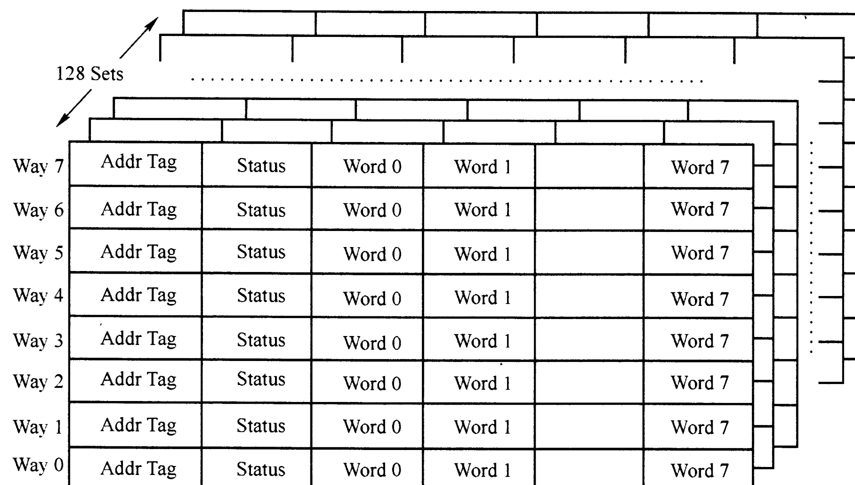


图 3-4　E500 L1 Cache 结构图

段降低一位。当路数所占位数为零时表示 Cache 使用直接映射的方式，当组数所占位数为零时表示 Cache 使用全相联结构。

E500 内核的 32 位地址与 L1 Cache 进行映射时，其地址可以分解为 Tag Address、Set Number、Word Select 和 Byte Select 四个字段，如下所示：

第 0～19 位	第 20～26 位	第 27～29 位	第 30～31 位
Address Tag	Set Number	Word Select	Byte Select

E500 内核的 32 位有效地址中 Set Number 字段占用地址的第 20～26 位，其长度与 L1 Cache 中的 Set Number 个数相同；Word Select 字段占用地址的第 27～29 位，用来选择在同一个 Cache 行中的 8 个 Word；Byte Select 字段占用地址的第 30～31 位，用来选择在一个 Word 中的某个字节；Address Tag 字段占用地址的第 0～19 位，该字段用来与 L1 Cache 中的 Addr Tag 进行比较，其长度与 Cache 中的 Addr Tag 长度相同。

E500 内核的 L1 Cache 行长度为 32B，组数为 128，路数为 8。因此处理器对 L1 Cache 进行查询时首先使用地址的 20～26 位在 L1 Cache 的 128 个组中进行组选择，然后将地址中存放的 Address Tag 与在同组的 8 路 Cache 行中的 Addr Tag 同时进行比较，如果其中有一路命中则根据地址的第 27～31 位在 Cache 行中选择需要的 Word，如果不命中则表示 L1 Cache 没有缓存所需数据，处理器将对 L2 Cache 和内存依次进行访问以获得所需要的数据。

由此可见，如果系统的 Cache 使用 16 路，32 路或者更多的路数，存放在 32 位地址中的 Address Tag 需要与 16 路或者 32 路 Cache 行中的 Addr Tag 进行比较，此时进行 Cache 设计，将需要更多的 CAM 存储器进行这种地址比较。

3.2.2　L1 Cache 的替换算法

E500 内核中 L1 Cache 的大小远小于系统中内存的大小，因此 E500 内核将不可避免地需要对 L1 Cache 中的 Cache 行进行替换。常用的 Cache 替换算法有 LRU 和 PLRU。

LRU(Least Recently Used)算法的原理较为简单，即淘汰最近没有使用的数据，对于采用 8 路组相联的 L1 Cache，采用这种算法将淘汰 8 路中的最近没有使用的 1 路 Cache 行。但是在

E500 内核中没有采用 LRU 算法而是采用 PLRU(Pseudo LRU)算法。在绝大多数的处理器中都使用 PLRU 算法而不是 LRU 算法实现 Cache 的替换。

E500 内核使用 PLRU 算法对 8 路组相联结构的 L1 Cache 进行替换,为此 E500 内核在 L1 Cache 中设置了 B[0～6]字段与 8 路 Cache 进行对应,如表 3-5 所示。

表 3-5　PLRU 算法中被淘汰的路与 B[0～6]之间的关系

<table>
<tr><th colspan="6">PLRU 字段</th><th>淘汰的路</th></tr>
<tr><td rowspan="8">B0</td><td>0</td><td rowspan="4">B1</td><td>0</td><td rowspan="2">B3</td><td>0</td><td>L0</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>L1</td></tr>
<tr><td>0</td><td>1</td><td rowspan="2">B4</td><td>0</td><td>L2</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>L3</td></tr>
<tr><td>1</td><td rowspan="4">B2</td><td>0</td><td rowspan="2">B5</td><td>0</td><td>L4</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>L5</td></tr>
<tr><td>1</td><td>1</td><td rowspan="2">B6</td><td>0</td><td>L6</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>L7</td></tr>
</table>

使用 PLRU 算法进行 Cache 替换需要遵循以下原则:

- 在系统初始化结束后,B0～6 位都为 0。L1 Cache 中所有 Cache 行的状态为 Invalid。
- 使用 PLRU 算法时,优先使用状态为 Invalid 的 Cache 行,E500 内核将按照 0～7 的顺序依次替换状态为 Invalid 的 Cache 行。在 E500 内核复位后,首先选择控制逻辑选择 L0 进行 Cache 行替换。
- 如果所有的路中 Cache 行的状态为 Valid,Cache 控制逻辑采用 PLRU 算法对 Cache 行进行替换。
- 当使用 PLRU 算法选定的 Cache 行的状态为 Locked 时,控制逻辑不能替换此 Cache 行,而是选择同组的其他路 Cache 行进行替换。如果同组的 8 路 Cache 行的状态都为 Locked,此时将启动外部总线周期从外部存储器中获得相应的数据。

PLRU 算法与 LRU 算法相比较为复杂。在初始化时,B0 与 B6 位都为 0。由上表所示,此时 B0,B1 和 B3 为 E500 内核将淘汰 L0,同时将 B0,B1 和 B3 位取反,即将 B0,B1 和 B3 位取反;此时 B0 为 1,B2 为 0,B5 为 0,此时 E500 内核将淘汰 L4,然后再将第 B0,B2 和 B5 位取反,并依此类推,形成一个状态机,然后对 Cache 中的 8 路进行替换。

值得注意的是,采用 PLRU 算法对 32 KB 的数据 Cache 进行刷新时,需要使用 dcbf 指令对 48KB 而不是 32KB 的连续数据区域进行操作。表 3-6 中简单列出使用 PLRU 算法对一个地址连续空间进行操作时,B0 位到 B6 位的状态转化以及如何进行 Cache 替换。

表 3-6　B0～B6 的状态转化表

B0	B1	B2	B3	B4	B5	B6	淘汰的路
0	0	0	0	0	0	0	L0
1	1	0	1	0	0	0	L4
0	0	1	1	0	1	0	L1
1	1	1	0	0	1	0	L6

(续)

B0	B1	B2	B3	B4	B5	B6	淘汰的路
0	1	0	0	0	1	1	L2
1	0	0	0	1	1	1	L5
0	0	1	0	1	0	1	L3
1	1	1	0	0	0	1	L7
0	1	0	0	0	0	0	L2
1	0	0	0	1	0	0	L4
0	0	1	0	1	1	0	L0
1	1	1	1	1	1	0	L6
...							

对此有兴趣的读者可以将B0位到B6位在进行Cache淘汰时的状态变换继续列下去。如果有可能,可以写一个程序将这一序列穷举出来。

从这一序列中我们可以发现,采用PLRU算法进行Cache淘汰时,将L0～L7路淘汰完毕,仅仅对8个Cache行进行操作是不够的。如表3-6所示,我们在其中任意挑选8个状态时,可以发现在8个状态中有重复的状态。在最糟糕的情况下,连续12个状态才能够包含L0～L7所有的状态。

因此,至少需要对12个Cache行进行操作才能完成对8路Cache行的替换。因此使用PLRU算法需要对48KB地址连续的空间进行操作才能完成对32 KB大小的Cache进行刷新。希望读者对此能够认真理会。实际上从数学上可以证明这一结论,对此有兴趣的读者,可以对16路、32路或者更多路组相联的Cache证明使用PLUR算法需要对多大的地址连续空间进行访问,以完成Cache的刷新。

许多处理器在对8路及以上的Cache进行刷新时,都采用了PLRU而不是LRU算法。其原因主要是PLRU算法虽然相对较为复杂,但是对于IC工程师PLRU算法的实现效率远高于LRU算法。

使用LRU算法时,不可避免地需要对8路Cache都进行比较以选择出最近没有使用的Cache行。这种比较算法将带来较大的处理延时,这种延时对于机器周期只有1ns或者几百ps的处理器而言,是非常大的。为此绝大多数处理器都使用了PLRU算法进行多路组相联结构的Cache行替换。

事实上,有过处理器内核设计经验的读者一定了解,在处理器内核设计中处处都会遇到瓶颈,处理器的频率提高也会带来许多问题。处理器采用的一些看起来并不算完善的算法,实际上是不得已而为之。

3.2.3 L1 Cache的状态位与L1 Cache的一致性

E500内核使用MESI协议对数据L1 Cache进行管理,MESI协议可以处理单处理器内核的L1 Cache,L2 Cache和主存储器间的一致性,也可以处理多处理器内核间的Cache共享一致性。目前基于使用总线监听法的MESI协议是处理Cache一致性最流行的协议。MESI协议规定每个数据Cache行中都包含MESI四个状态位。此外,E500内核在L1数据Cache中还

支持一个 Lock 位。在单处理器的体系结构下 MESI 的定义如下所示：

（1）M(Modified)。M 位为 1 时表示当前 Cache 行中包含的数据不在本处理器内核的内存体系中，此时 Cache 行中所含的数据无效。当处理器对此 Cache 行进行操作时，必然会导致 Cache 失效，从而引发系统总线的读写周期，将 Cache 行中数据与内存中的数据同步。

（2）E(Exclusive)。E 位为 1 时表示当前 Cache 行中包含的数据仅在本处理器内核的内存体系中。在单处理器系统中，如果 Cache 行命中时，此 Cache 行的状态为 Exclusive。

（3）S(Shared)。在单处理器结构中，Cache 行中的状态不可能为 Shared。在多处理器系统中，Cache 行的状态可能为 Shared。

（4）I(Invalid)。该位用来表示当前 Cache 行是否有效，在 E500 内核使用 PLRU 算法对 Cache 行进行替换时，将首先替换状态为 Invalid 的 Cache 行。

在指令 L1 Cache 中只有此位有效，在计算机体系结构中指令 Cache 的处理较为简单，指令 Cache 不存在一致性的问题，因为在现代计算机体系结构中，程序的正文段一般不被修改。如果用户由于某种特殊需要对程序的指令段进行修改时，需要编写 Cache 同步的指令，维护程序地址空间与指令 L1 Cache 之间的同步。这种对程序正文段进行修改的程序，也被称为 self-modifying 程序。

（5）Locked。该位表示当前 Cache 行是否被加锁。被加锁的 Cache 行不能被替换。有的程序将某些经常使用的程序或者数据在 Cache 中锁定以提高程序的效率。

但是将 Cache 锁定的做法并不一定能提高程序的整体效率。程序员可以根据具体情况对这些现象进行分析。除非万不得已，不赞成程序员将常用的数据或者程序锁定到 L1 Cache 中，因为这种做法将极大地影响程序的可移植性。

单处理器系统在进行 Cache 共享一致时，只使用了 MESI 协议的子集。此时 Cache 的共享一致协议比较容易实现。

1．基于单处理器 Cache 的存储器读操作

E500 内核在进行存储器读操作时，首先检查当前访问的数据是否在 L1 Cache 中命中。

如果访问的数据在 L1 Cache 中命中，则进一步检查 L1 Cache 行的状态。如果 Cache 行中的数据有效，即 Exclusive 位为 1，则将从 L1 Cache 中获得相应的数据；如果 Cache 行中的数据无效，即 Modified 位为 1，则以 Cache 行为单位对本处理器的内存系统进行读操作获得相应的数据，同时更新此 Cache 行中的数据并将 Cache 行状态的 Exclusice 位更改为 1，表示当前 Cache 行中的数据有效。

如果访问的数据没有在 L1 Cache 中命中，则使用 PLRU 算法计算出需要淘汰哪一个 Cache 行，然后根据 Cache 行的状态决定如何淘汰该 Cache 行。如果该 Cache 行状态的 Exclusive 位为 1，则直接将淘汰此 Cache 行中的数据，不需要与存储器系统进行同步；如果该 Cache 行状态的 Modified 位为 1，则需要将在 Cache 行中的数据与存储器同步。之后将启动存储器读周期，将数据从 L2 Cache 或者内存中读入到该 Cache 行中，同时将此 Cache 行状态的 Exclusive 位更改为 1。

2．基于单处理器 Cache 的存储器写操作

E500 内核在进行存储器写操作时，可以使用两种策略，回写（Write-Back）或者通写（Write-Through），对 L1 Cache 进行更新。下文将分别介绍采用回写和通写两种情况进行存储器写操作时，L1 Cache 行状态的转换。

(1) 使用回写策略。此时 E500 内核在进行存储器写操作时,首先检查当前访问的数据是否在 L1 Cache 中命中。如果数据在 L1 Cache 中命中,则将数据写入 L1 Cache 中,此时不改变 L1 Cache 行的状态。

如果数据没有在 L1 Cache 命中,则使用 PLRU 算法计算出需要淘汰哪一个 Cache 行,然后根据 Cache 行的状态决定如何淘汰该 Cache 行。淘汰 Cache 行的方法与 3.2.3 节的淘汰方法相同。将 Cache 行的数据淘汰后,将新的数据写入此 Cache 行。此时 Cache 行的状态为 Modified。

(2) 使用通写策略。此时 E500 内核在进行存储器写操作时,首先检查当前访问的数据是否在 L1 Cache 中命中。如果数据在 L1 Cache 中命中,则将数据写入 L1 Cache 中,同时将此数据写入到内存体系中,并将该 L1 Cache 行状态的 Exclusice 位置为 1。

如果数据没有在 L1 Cache 中命中,则使用 PLRU 算法计算出需要淘汰哪一个 Cache 行,然后根据 Cache 行的状态决定如何淘汰该 Cache 行。淘汰 Cache 行的方法与 3.2.3 节的淘汰方法相同。将 Cache 行的数据淘汰后,将新的数据写入此 Cache 行,同时将数据写入内存体系中,并将该 L1 Cache 行状态的 Exclusice 位置为 1。

如果在一个处理器系统中,只使用通写策略进行 L1 Cache 的更新,Cache 行的状态不可能为 Modified。如果在一个系统中同时采用了通写和回写两种策略,并将两个分别采用回写策略和通写策略的虚拟地址映射到同一个物理地址时,Cache 行的状态就有可能为 Modified。

使用通写方式时,E500 内核对系统总线的访问频率较高,从而会在一定程度上加大系统总线的负担。因此在多数应用中,使用回写策略管理 L1 Cache。但是共享数据的管理一般要使用通写策略,以避免 Cache 的颠簸。

3.2.4 与 L1 Cache 管理有关的寄存器

E500 内核设置了一些寄存器管理 L1 Cache。这些寄存器包括 L1CSR0, L1CFG0, L1CSR1 和 L1CFG1 寄存器。这些寄存器的主要作用是对 L1 Cache 进行管理和配置。

L1CSR0(L1 Cache Control and Status Register 0)寄存器。该寄存器用来对数据 L1 Cache 进行设置,其字段如下所示:

- CPE 位。该位为 1 时,使能数据 L1 Cache 的奇偶校验检查,此时如果发生了 Cache 的奇偶校验错误,E500 内核将产生 Machine Check 异常。
- CPI 位。该位置 1 时,将翻转当前数据 L1 Cache 的奇偶位,从而将引发 Machine Check 异常。设置该位的主要目的是用来调试系统的异常处理软件。只有当 CPE 位为 1 时,CPI 位才能被置为 1。
- CSLC, CUL, CLO 和 CLFR 位被称为 Line Locking APU 位。该四位用来反映使用指令 dcbi 是否能对 Cache 行解除锁定。
- CFI 位。对此位写 1 将使无效数据 L1 Cache,当 E500 内核使无效 Cache 操作完成后将此位自动清零。
- CE 位。此位将使能或者关闭数据 L1 Cache。

L1CFG0(L1 Cache Configuration Register 0)寄存器。该寄存器用来记录数据 L1 Cache 中的配置信息,该寄存器只读。

- CARCH 字段。该字段为 00 表示当前 Cache 使用 Harvard 结构。

- CBSIZE 字段。该字段为 00 表示 Cache 行长度为 32 B,256b;为 01 表示 Cache 行长度为 64 B,512b。
- CREPL 字段。该位为 00 表示 Cache 的替换算法为 LRU,为 01 表示 Cache 的替换算法为 PLRU。
- CLA 位。该位为 0 表示 Line Locking APU 位在系统中无效,为 1 表示 Line Locking APU 位在系统中有效。
- CPA 位。该位为 0 表示 Cache 中没有奇偶校验功能,为 1 表示 Cache 中具有奇偶校验功能。
- CNWAY 字段。该字段为 111 时,Cache 采用 8 路组相联结构。
- CSIZE 字段。该字段用来表示 Cache 的大小,该字段为 0x20 表示当前 Cache 的大小为 32 KB。

E500 内核中还有 L1CSR1 和 L1CFG1 寄存器,与 L1CSR0 和 L1CFG1 寄存器中不同,这组寄存器的主要作用是对指令 L1 Cache 进行操作。这两个寄存器中的字段与 L1CSR0 和 L1CFG1 寄存器中的字段相似。

3.2.5 与 L1 Cache 管理有关的指令

E500 内核中设置了许多指令对 Cache 进行操作。

- dcba 指令。该指令的格式为"dcba rA, rB"。此指令为有效地址 EA 分配一个 Cache 行,然后此 Cache 行清零。如果此 Cache 行状态为 Modified,则需要将此 Cache 行回写到内存,然后再将此 Cache 行清零。
- dcbf 指令。该指令的格式为"dcbf rA, rB"。此指令将有效地址 EA 所在的 Cache 行刷新,当此 EA 没有在数据 L1 Cache 中命中时,此指令直接返回不做任何操作。当在数据 L1 Cache 中命中,则根据命中的 Cache 行状态进行相应操作。如果该 Cache 行的状态为 Modified,则将 Cache 行中的数据写入内存,然后将状态置为 Invalid;如果为其他状态,则直接将此 Cache 行状态置为 Invalid。
- dcbi 指令。该指令的格式为"dcbi rA, rB"。此指令将有效地址 EA 所在的 Cache 行置无效。与 dcbf 指令不同,dcbi 直接将命中的 Cache 行的状态置为 Invalid,而不将状态为 Modified 的 Cache 行回写到内存。此指令要求处理器必须在超级用户模式才能执行。
- dcbz 指令。该指令的格式为"dcbz rA, rB"。此指令与 dcba 指令类似。只是此指令会对 Cache 的状态进行检查。当 Cache 行不使能,Cache 行被锁时或者 Cache 处于其他不能被改写的状态,执行"dcbz"指令会引起 alignment 的中断。
- dcbst 指令。该指令的格式为"dcbst rA, rB"。此指令将有效地址 EA 所在的 Cache 行与内存同步。当此 EA 没有在数据 L1 Cache 中命中时,此指令直接返回,不做任何操作。当在数据 L1 Cache 中命中,则根据命中的 Cache 行状态进行相应操作。如果该 Cache行的状态为 Modified,则将 Cache 行中的数据写入内存,然后将状态置为 Exclusive;如果为其他状态,该指令不做任何操作。
- dcbt:指令格式为"dcbt CT, rA, rB"。CT 字段为 0 表示是对数据 L1 Cache 操作,为 1 表示对数据 L2 Cache 进行操作。此指令将地址 EA 中的数据预取到相应的数据 Cache 行中。

- dcbtst:指令格式为“dcbtst CT, rA, rB”。CT 字段的含义同上。dcbtst 指令与 dcbt 指令类似,也是将内存的数据填充到指定的 Cache 行中。如果 dcbtst 指令所要访问的数据在数据 L1 Cache 中,此指令基本不用做任何操作;如果 dcbtst 指令所要访问的数据在数据 L2 cache 中命中而在数据 L1 Cache 中失效,则此指令将在 L2 中的数据调入数据 L1 Cache 中;如果 dcbtst 指令所要访问的数据在数据 L1 和 L2 cache 中均不命中,则此指令将在内存中的数据调入数据 L1,L2 Cache 中。
- dcblc 指令。该指令的格式为“dcblc CT, rA, rB”。CT 字段的含义同上。此指令将有效地址 EA 所在的 Cache 行的 Locked 位清除。
- dcbtls 指令。该指令的格式为“dcbtls CT, rA, rB”。CT 字段的含义同上。此指令将有效地址 EA 中存放的数据加载到相应的 Cache 行中,并将 Locked 位置 1。
- icbi 指令。该指令的格式为“icbi rA, rB”。此指令将有效地址 EA 所在的 Cache 行置无效。该指令用来操作指令 Cache。
- icbt 指令。该指令的格式为“icbt CT, rA, rB”。CT 字段的含义同上。此指令将有效地址 EA 所在的 Cache 行进行填充,将在内存中数据填充到指定的 Cache 行中。该指令用来操作指令 Cache。
- icblc 指令。该指令的格式为“icblc CT, rA, rB”。CT 字段的含义同上。此指令将有效地址 EA 所在的 Cache 行 Locked 位清除。该指令用来操作指令 Cache。
- icbtls 指令。该指令的格式为“icbtls CT, rA, rB”。CT 字段的含义同上。此指令将有效地址 EA 中存放的数据加载到相应的 Cache 行中,并将 Locked 位置 1。该指令用来操作指令 Cache。

上文仅仅简单解释了对 Cache 进行操作的指令,这些解释实际上只是针对单处理器内核。对于多处理器内核而言,这些指令的实现远比以上内容复杂得多。

3.3 E500 内核的存储器一致与同步

PowerPC 处理器使用弱序(Weakly order)对存储器进行访问。所谓弱序是一种暂时打破正常的存储访问一致性以提高整体系统性能的一种方法。

弱序与 CPU 的乱序执行(out of order)、条件转移指令的预测及指令流水线调度紧密相连。现代 CPU 中多采用流水线多发射技术,即 CPU 可以同时执行多条指令,包括运算、访问存储器指令等。在 PowerPC 处理器中,流水线的多发射,CPU 的乱序执行与条件语句的动态预测使得弱序存储器访问成为可能。

弱序存储器访问是指在乱序的指令执行中提前执行存储器访问的指令以缓解存储器的瓶颈问题的一种存储器访问方式。弱序存储器访问的实现进一步提高了对存储器带宽和流水线的有效利用,同时也带来了一些有关存储器一致性的问题。

在弱序访问机制下,如何恢复无效的存储器操作,如何对某些存储操作进行保护,需要流水线控制部件和操作系统协同完成。

系统程序员在书写驱动程序或 BSP 的代码时,必须考虑因为弱序存储器访问而带来的同步问题。PowerPC 处理器在 MMU 中提供了一些寄存器位和同步指令支持弱序存储器访问。

3.3.1 弱序存储结构的存储器分类

目前,学术界已对存储器的各种分类进行了比较透彻的分类,但从某些程序员的角度看,这些分类并不完全适用,起码现有的存储器分类和他们编程的难度并没有太多关系。

弱序存储访问要求程序员必须对不同种类的存储器选择不同的处理。对于一些编写设备驱动程序的工程师,必须注意 PowerPC 处理器的弱序存储器访问。本书根据 PowerPC 处理器的弱序存储器机制对存储器重新进行分类,如下所示:

(1) 静态存储器。静态存储器包括 SRAM,SSRAM 等等。对此类存储器,程序员只需简单地配置处理器内的一些寄存器,就可以由 CPU 对此类存储器自动进行访问。这类存储器经常被用作处理器系统的片外 Cache。目前在一些应用上,使用此类存储器的机会越来越少。越来越多的处理器内部包含了大容量的 L2 Cache,这些 L2 Cache 直接使用快速的系统总线时钟与 L1 Cache 进行通信,其效率远比外部的 Cache 高。一些简单的处理器,如 8/16 位单片机,还在继续使用此类存储器作为主存储器,用来保存程序及程序所使用的数据。

(2) 主存储器。主存储器包括 SDRAM,DDR SDRAM 等。所有的处理器都会重点考虑与主存储器的通信效率与带宽。为此,各类存储访问技术层出不穷,时钟频率越来越高,对硬件设计的要求越来越严格。

不夸张地说,现在的硬件工程师在设计高端应用时考虑最多的问题就是信号完整性,从投板到调试阶段,始终在担心因为 DDR 的时序带来的严重结果。因此在设计阶段,硬件工程师基本上都对 DDR 时序进行仿真模拟,遵循着各类由主流半导体公司提供的设计规范。

软件工程师配置 PowerPC 处理器的主存储器较为容易。PQIII 处理器包含 DDR 控制器,并提供了几个寄存器供程序员使用,对这些寄存器的配置需要软件工程师了解一些 DDR 的知识,有些寄存器的配置对于系统效率有较大影响。主存储器可以支持随机访问,存储空间连续,与 Cache 结合紧密,访问速度快。PowerPC 处理器支持的弱序访问也主要针对此类存储器。

(3) Flash 类存储器。Flash 类存储器包括 NOR-Flash、NAND-FLASH、CF 卡等。对存储容量有较大需求的应用,经常使用 NAND-FLASH 与 CF 卡进行数据储存。

NAND-FLASH 设备的访问速度较慢,出现数据坏块时无法修复,不过这一类 Flash 的容量可以很大,因此在移动存储、数码相机、U 盘等一些对容量需求较大的应用中得到了广泛的普及。

对于软件工程师,实现对此类存储器的支持不过是移植相应的驱动程序而已。需要注意的是许多设备商提供的驱动程序并没有考虑 PowerPC 处理器的弱序。因此程序员在移植此类驱动程序时要注意有关存储一致性的问题。

NOR-Flash 是工程师们最常用的 Flash 类型,这类 Flash 用来存放 Boot Loader 程序和操作系统映像。对于程序员,此类 Flash 的特点是进行写操作前必须进行擦除操作。此类 Flash 需要进行一系列命令字操作后才能进行擦除或写操作,具体的命令字结构请参考相应 Flash 的数据手册。无论是 NOR-Flash 还是 NAND-Flash 都有一个共同的特点,就是在进行数据写和擦除时,需要完成一个写序列,而且这个序列不能被中断。因此对这类存储器进行写操作时,类似于访问一种特殊的设备。

(4) 存储器映射的外设。存储器映射的外设包括 UART,网口,RapidIO,PCI Express 及

处理器内部的寄存器。此类设备是 PowerPC 软件工程师书写设备驱动程序的工作重点，这些外设也是 PowerPC 弱序存储器结构影响最为严重的内存区域。

此类存储器的读取与写入没有什么联系，程序员往往不能使用对同一地址的读取操作获得刚刚写入的数值。

与这些外设进行数据交换需要一个完整的操作序列来实现，而这一序列往往不能错序也不能被中途打断，如插入其他访问此类寄存器的指令。因此要求程序员对此类存储器进行保护使之不受到 PowerPC 弱序存储器访问机制的影响。

3.3.2 弱序存储器访问机制

弱序访问机制是与强序(Strong Ordering)相对而言的。强序机制是指程序对存储器的访问严格按照程序的执行顺序进行，而弱序机制是指对存储器的访问不一定按照程序的规定顺序进行。

在 E500 内核中，对存储器的写操作一定会按照程序的执行顺序进行，而读操作可以乱序执行，在 E500 内核中将这种乱序操作称为 Reordered Loads/Stores。在 E500 内核中，系统总线有两条数据读总线和一条数据写总线用以支持读操作的乱序执行。

图 3-5 中的程序在 E500 内核中执行时，其访问顺序有可能发生以下乱序：

- Load B 指令可以先于 Load A(Load/Load 乱序)。
- Store C 指令可以先于 Load B(Load/Store 乱序)。
- Load D 指令可以先于 Store C(Store/Load 乱序)。

这里再次提醒读者注意，E500 内核并不支持 Store/Store 类的乱序执行。实际上，没有哪一种处理器支持 Store/Store 类乱序，其原因不言而喻，在一个处理器中，可以轻易地将存储器读操作的结果丢弃，但是任何一个处理器系统都很难去除存储器写操作的结果。

如果在乱序执行中出现任何异常或者中断时，比如 Load D 指令在 Store C 指令之前执行，而在执行 Store C 时出现异常，进入异常处理程序。此时处理器将进行指令同步操作并且丢弃 Load D 指令的执行结果，而不影响程序的执行。在 E500 内核中除了支持存储器访问的乱序执行，还支持存储器的猜测访问(Speculative Access)。存储器猜测访问的例子如图 3-6 所示。

Load A
Load B
Store C
Load D
Store E
Store F
Load G

图 3-5　存储器访问的例子

Loop:Load A
...
ADDI
BC Loop
Load B
SUB
Store C

图 3-6　猜测访问的例子

而在 PowerPC 的弱序访问机制下，Load B 指令可以在 BC Loop 指令执行执行完毕之前运行，这种在转移指令之前进行的存储器访问操作被称为猜测访问。此时，如果 BC Loop 指令进行跳转并执行 Load A 指令，CPU 将放弃本次 Load B 的读取结果；如果 BC Loop 指令不进行

跳转,则 CPU 保留 Load B 指令的执行结果而去执行 ADDI 指令。

采用此方法可以提高处理器的指令流水线效率及对存储器带宽的利用。PowerPC 体系结构中的猜测访问并不支持存储器写操作,同时也不能对被保护的存储器(Guarded Memory)进行猜测访问。

以上的猜测访问和乱序操作对主存储器和静态存储器并不会产生影响,因为对这些存储器进行读操作并不会影响这些存储器的状态,E500 内核将这类存储器称为 Well-behaved 存储器。但是猜测访问会对 Flash 类存储器和存储器映射的外设带来很大的副作用,在这一类存储器中,如果进行猜测读访问有可能会打破对这类存储器操作的序列,从而带来一些不可预料的结果。

为支持 PowerPC 体系结构的存储器乱序和猜测访问, PowerPC 体系结构在 MMU 中增加了一些特殊的位以及几个同步指令以支持 PowerPC 的弱序访问机制,从而维护 PowerPC 体系结构的存储器访问一致性。

3.3.3 PowerPC 处理器的存储器访问一致性

存储器访问一致性模型要求整个系统提供一个统一的存储器映像,以便对散布在不同处理器中的数据进行共享。但是有时共享数据的拷贝会出现在 Cache 里或者主存储器中,从而有可能引发由于数据不同步造成的错误。对于多处理器结构,如 SMP,这种错误更容易发生,也要费更大的代价来处理数据同步的问题。本节将重点介绍单处理器中的存储器访问一致性。

1. WIMGE 字段

WIMGE 字段是 E500 内核进行虚实地址转换时,用来描述物理地址访问属性的字段。如 3.3.1 节所示,E500 内核中支持两类 TLB,即 TLB0 与 TLB1。这两种 TLB 在进行虚实地址转换时都使用了 WIMG 四位用来处理数据的同步问题。

(1) W(Write-through)位。如果此位为 1 表示对此数据区域的访问采用 Write-through 的 Cache 策略,如果为 0 表示使用 Write-back 的策略。

(2) I(Caching-inhibited)位。如果此位为 1 表示对此数据区域访问时不使用 L1 Cache 进行数据访问优化,如果为 0 表示使用 L1 Cache。

(3) M(Memory-Coherence-Required)位。如果此位为 1 表示对此数据区域的访问需要进行存储一致性处理。此位对于单 PowerPC 系统而言没有什么意义。对于 SMP 结构的处理器系统,由 Memory Coherence 引起的 Cache 共享一致性的问题十分复杂。在 SMP 系统中,维护 Cache 一致性的方法较多。如写更新法,在写操作完成之后,其数据在其他 Cache 中无效;读请求法,如果 Cache 里出现一个脏拷贝,则需要回写操作在读之前完成,当然这些方式需要硬件支持完成,如基于总线监听法保证 SMP 结构处理器的 Cache 的一致性。

在 SMP 结构处理器中,有些内存区域需要在多个处理器中共享,此时在使用 MMU 进行虚实地址映射时将 WIMG 字段的 M 位置 1。此时处理器对该段数据的进行访问时一定会通知在 SMP 系统中的其他处理器,以实现存储器的共享一致。

目前还没有基于 E500 内核的多处理器系统(Freescale 将会在近期发布第一个基于 E500 的双核处理器 MPC8572)。此时程序员需要将此位置为 0,从原理上说,此时将此位置为 1 也不会引发任何错误,但是将 M 位置为 1 这种做法可能并没有得到大规模、有强度的测试。

(4) G(Guarded)位。此位为1表示对相应的存储区域进行保护。这一设置可以用来阻止PowerPC处理器的猜测访问。Flash类和外设类的存储区域需要在程序中设置此位，以阻止猜测访问。比如E500内核进行Flash擦除操作时，有可能会出现与图3-6中类似的算法。例如进行Flash擦除操作时，需要首先写入某些命令字序列，之后循环对某个状态位进行测试。此时如果允许猜测访问，将有可能破坏对Flash擦除的指令序列，从而导致Flash擦除的失败。

对于其他外设的访问有可能也会出现这种类似情况。因此对于Flash类和外设类的存储区域进行虚实地址转换时，需要对G位进行设置。

除了这四位之外，E500内核还提供了msync，mbar，isync，lwarx，stwcx. 等其他同步指令用于指令与存储器同步。

(5) E(Endian)位。该位为1时，表示对此数据区域的访问采用小端模式。PowerPC处理器一般使用大端方式。

2. 存储器同步指令

在E500内核中，对某些寄存器进行访问(如MSR，HID1，L1CSR0寄存器)及执行某些对TLB进行操作的指令(tlbivax和tlbwe)也需要使用存储器同步指令，如msync指令或者mbar指令，进行同步处理。对MSR寄存器的WE位进行操作时，PowerPC处理器将进入各种低功耗状态，此时PowerPC处理器要求在对此位操作之前使用msync指令。此外对HID1和L1CSR0寄存器的某些位进行操作时也需要使用msync指令进行同步。

在执行tlbivax指令或者tlbwe指令后会影响相应TLB的Entry，如果后续指令对存储器的访问将使用这些Entry，就一定要使用msync指令进行同步，以确保这些存储器访问的正确执行。

在E500体系中，对存储器的访问一般要使用lw或st类指令，而设置特殊寄存器一般使用mtspr指令。从CPU的执行中这两类指令不会产生任何相关，但在实际执行中，对存储器的访问操作必须要等待这些特殊寄存器的设置完毕后才能进行。此时在这些mtspr指令后必须要执行msync指令以保证存储器访问的正确执行。

产生这些存储器同步指令的根本原因是由于PowerPC处理器支持的乱序访问和猜测访问而带来的存储器相关问题。本书将这类相关称为“存储器隐性相关问题”。

PowerPC处理器在与许多外设进行数据交互时，必须遵循一些特定的访问序列，有时这些访问序列也带来了“隐性相关问题”。比如在一个外部设备中存在两个寄存器UD-R1和UD-R2，其中UD-R1可以进行写操作，UD-R2可以进行读操作，PowerPC对UD-R1进行写操作时会影响到UD-R2的值，这时也带来一些“隐性相关问题”。

PowerPC处理器和许多支持多发射、存储器乱序执行的处理器都会遇到这些“存储隐性相关问题”。PowerPC处理器采用了增加相关同步指令来处理这些“存储隐性相关问题”，有些这类相关问题可以使用硬件的方法进行，不过这样的设计过于复杂，耗费的资源过多。

承担内核IC设计的系统工程师被乱序执行带来的各种相关问题困扰着，实在没有更多的精力处理这些“隐性相关问题”。除此之外，PowerPC还要与许多外设进行连接，这些外设有些是自定义的，对于这种自定义外设的存储器进行访问而造成的“存储隐性相关问题”更是无法解决，因此PowerPC处理器设立了一些存储同步指令以保证整个系统的同步，如msync和mbar指令。

这两条存储同步指令也被称为存储器栏栅指令(Memory Barrier或Memory Fence)。这类

指令要求在其之前执行的对存储器的操作必须要执行完毕才能执行这类指令之后的存储器读写指令。存储器栏栅指令如其名称所示，类似一个栏栅将其前执行的存储器访问指令和在其后执行的存储器访问指令隔离开来。

其中E500内核的msync指令的执行码OPCD与其他PowerPC体系结构（如603E内核）的"sync"指令相同，其意义有些类似。msync指令用于确保在此语句之前的所有存储器访问语句结束后，再开始执行msync指令后的存储器访问语句。

执行msync指令会耗费处理器大量的时间用于存储器同步并会中断指令流水线的执行，频繁使用msync指令会极大地降低处理器效率，因为msync指令无论访问任何内存区域时（无论WIMG字段为何值时），都会建立一个存储器栏栅用来对存储器访问进行同步。因此E500内核还引入了一条mbar指令，mbar指令与603E内核的eieio指令的操作码OPCD完全相同，但含义略有不同。

eieio指令的英文是Enforce In-Order Execution of I/O，在许多PowerPC体系结构中将此指令实现为"No-op"（不做任何操作）指令。eieio指令对不使能Cache的存储器区域（WIMG字段的I位为1）进行同步，其作用也是对存储器操作指令间设置栏栅，但是eieio指令只对不使能Cache的数据区域有效。对于一些PowerPC内核，由于对不使能Cache的数据区域的只能进行强序访问，因此eieio指令在此类PowerPC内核中无效。

mbar指令具有一个操作数MO，其指令的格式为"mbar MO"。其中MO是一个5b大小的字段，取值范围为0～31。

- 在E500内核中。当MO ≠ 1时，"mbar MO"指令与msync指令相同。
- 当MO = 1时，"mbar MO"指令可以在两种情况下充当存储器栏栅指令，一种与其他PowerPC体系下的eieio指令的作用完全相同；而另一种情况是针对多处理器结构的，当访问内存区域的WIMG字段中的W位为0，I位为0而M位为1时将对存储器写操作进行同步。

由此可见，对于单处理器系统eieio等效于"mbar 1"指令，eieio指令实际上是mbar指令的一个子集，但是其执行效率要比msync指令高。在E500内核中不支持eieio指令，而是使用mbar指令实现eieio指令。

许多基于PowerPC处理器的操作系统对外设的写操作后都加入了eieio指令。如Linux系统中对外设的写函数out _ le32。

```
extern inline void out _ le32(volatile unsigned __ iomem * addr, int val)
{
__ asm __ __ volatile __("stwbrx %1,0,%2; eieio" : "=m" ( * addr) :
        "r" (val), "r" (addr));
}
```

本书建议在进行嵌入式系统开发时将外设的写操作与"eieio"指令进行封装，并不建议程序员直接使用stw类指令对外部设备直接进行访问。

3. 指令同步

在PowerPC E500中有两类指令可以用作指令同步，isync和sc指令以及rfi类指令，rfi类指令包括rfi，rfci和rfmi指令。

指令同步与之前提到的存储器同步指令没有直接联系。这类指令的主要作用是排空指令

流水线。排空指令流水线包含两方面内容，一是将未执行的指令从指令队列中抛弃，二是等待已执行的指令结束。因此在指令同步之后，处理器将重新对指令进行预取。

isync 指令是一个显式同步指令。PowerPC 处理器规定了在执行一些指令后必须使用 isync 指令进行同步。这些需要在其后加入 isync 的指令一般都有一个共性，即这些指令改变了其后指令的运行空间。

sc，rfi 类指令分别是系统调用(system call)指令和中断返回指令。执行这些指令会切换程序的执行空间，因此这些指令的执行也要求指令流水线的排空。对于一个中断和系统调用频繁的应用，sc 和 rfi 指令引起的流水线排空操作是导致系统效率低下的主要原因，而且处理器的指令流水线的深度越深，这种影响越大。

3.4 CCB 总线的设计

CCB 总线是 E500 内核的前端总线，E500 内核并没有公开其系统总线设计的细节。这条总线对绝大多数软件(包括系统软件)透明。基于 E500 内核的处理器并不像基于 603E 内核的处理器将 60X 总线引出，而是在片内使用这条总线。这样做的主要原因如下：

- 芯片内部前端总线的频率越来越高，将这条总线直接引到外部并保持与在 SOC 中同样的时钟频率比较难以实现。
- 对于绝大多数设计，没有必要向用户提供前端总线。前端总线包含了许多存储器一致和 Cache 同步的信号，一般用户无需使用这些信号。
- 前端总线一般在 SOC 的设计中才会用到，一般来说前端总线的引脚过多，在芯片设计时也没有太好的办法将前端总线全部引出到外部引脚。

E500 手册屏蔽了大多数 CCB 总线的实现细节，包括许多重要的硬件引脚。一般来说嵌入式处理器很少将前端总线完全引到片外。

CCB 总线继承了 60X 总线的大部分内容，并略有改进，而 60X 总线又是基于摩托罗拉的第一颗 RISC 芯片 88110 处理器的系统总线。从体系结构的角度上看 88110 处理器的系统总线，60X 总线和 E500 内核的 CCB 总线并没有质的变化。作为处理器的前端总线，CCB 总线具有以下特性：

- 可以连接多个处理器，支持 SMP 结构的多处理器。这些处理器可以使用总线监听法实现内部 Cache 的共享一致性。CCB 总线也可以支持 L2 Cache，CPM 和 SOC 中的的其他外设与 L1 Cache 的共享一致性。
- 可以支持 inline Cache 或者 look-aside 方式的 Cache。支持 Cache 和 TLB 管理的指令，许多 Cache 和 TLB 管理的指令需要广播到 CCB 总线上，以保证整个系统的 Cache 和 TLB 的一致性。
- 支持多处理器对主存储器的共享。
- 支持弱序存储器访问。
- 支持处理器对存储器的原子操作。

CCB 总线由总线仲裁信号、总线控制信号、地址总线与数据总线组成。CCB 总线含有两条独立的数据读总线，一条数据写总线用来支持 PowerPC 处理器对 CCB 总线的读写操作，CCB 总线为这三条总线提供了不同的控制与命令信号以保证处理器有可能同时操作这三条

总线。

CCB 总线中含有两条独立的地址总线，一条地址总线作为输出，可以用来对 CCB 总线上的设备进行访问，而另一条地址总线作为输入，可以支持 CCB 总线的监听。

3.4.1 CCB 总线访问周期

CCB 总线是一个同步总线，其地址总线和数据总线分离，可以支持地址流水和分离传送两种较为先进的数据传送方式。在多处理器系统中，这些总线传送方式可以极大地提高 CCB 总线的利用率。

CCB 总线的访问周期分为总线仲裁，数据传送和总线结束 3 个阶段。

(1) 总线仲裁阶段。在此阶段，总线仲裁器需要对 CCB 总线进行访问的设备进行仲裁，以确定设备可以使用哪条总线的使用权。CCB 总线仲裁器可以分别为数据总线和地址总线进行独立仲裁。

因此在 CCB 总线中的设备可以分别获得地址，数据总线的使用权或者同时获得地址和数据总线的使用权。这种总线仲裁方式使得 CCB 总线 Address pipelining 和 split transaction 数据传送成为可能。

(2) 数据传送阶段。CCB 总线的数据传送分为地址周期和数据传送周期。当一个外部设备获得地址总线使用权后，即可开始数据传送的地址周期。在进行数据传送时，设备完成地址总线的使用后将释放本次地址总线的使用权，此后总线仲裁器就可将地址总线分配给其他使用地址总线的设备，而无需等待数据传送完毕，这种数据传送方式有效地支持了 Address Pipelining 和 Split Transaction 数据传送方式。

而对某些访问延时较大的设备进行数据传送操作时，CCB 总线还支持 Split Transaction 数据传送方式，在这种方式下，一次 CCB 总线周期中可以插入其他对 CCB 总线的访问周期。Address Pipelining 和 Split Transaction 数据传送的时序如图 3-7 所示。

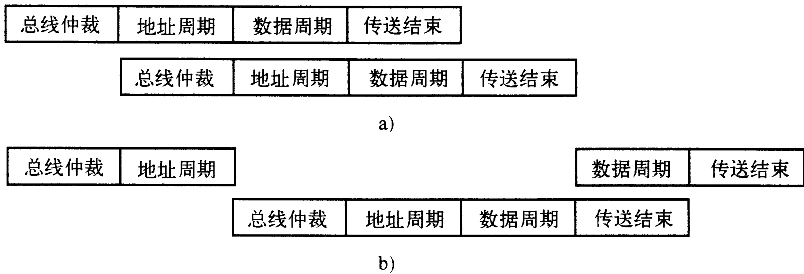


图 3-7　Address Pipelining 和 Split Transaction 数据传送方式

a) Address Pipelining 传送方式　b) Split Transaction 数据传送方式

(3) 传送结束阶段。在此阶段中，CCB 总线主设备停止对地址与数据总线的使用并获取传送成功或失败信息，结束 CCB 总线的数据传送周期。

3.4.2 CCB 总线的主要数据信号线

CCB 总线中包含的信号线非常多。本节只对 CCB 总线中的控制和命令信号进行说明，这些信号包括 tt[0:4]，gbl#，ci#，cl# 和 ts# 信号。

● tt 信号共有 5 位,用来描述 CCB 总线中定义的数据传送方式。
● glb#信号用来反映 WIMG 字段中的 M 位,当对 M 位为 1 的数据区域进行访问时,该信号有效,该信号用来处理 Cache 的共享一致。
● ci#信号用来反映 WIMG 字段中的 I 位,当对 I 位为 1 的数据区域进行访问时,该信号有效,该信号用来访问不使能 Cache 的数据区域。
● cl#信号用来反映 L2 Cache 的锁存位。
● ts#信号用来表示 CCB 总线的数据周期的开始。

3.4.3 CCB 总线操作

CCB 总线支持的总线操作方式大于 16 种,共使用 5 位状态信号 tt [0:4] 区分这些总线操作方式。CCB 的这些总线操作共分为三大类:读操作、写操作和地址访问操作方式。

其中 CCB 总线读写操作可以以 Cache 行为单位对 CCB 总线进行突发读写,也支持 1 个字节到 8 个字节的单次总线读写操作。CCB 总线的读写操作需要使用 CCB 总线的地址和数据总线;CCB 总线上的地址访问操作支持一些对 Cache,MMU 的操作,以及一些同步指令和一些地址广播操作。CCB 总线上的地址访问操作仅仅使用 CCB 总线的地址线,而不使用数据总线。

CCB 总线保证所有连接在这条总线上的 Cache 和主存储器的一致性。这种一致性基于总线监听法。CCB 总线要求所有需要支持存储一致的设备必须对 CCB 总线进行监听并根据监听的结果改变 Cache 行中的 MESI 状态以保证存储一致性。MPC85XX 芯片使用 SDMA 进行数据传送时,可以设置 GBL 位,当此位为 1 时,使用 SDMA 数据传送将保证 Cache 的一致性,这一功能显著地提高了 SDMA 的效率。

如果 CCB 总线不支持 SDMA 访问与 Cache 的一致,挂接在 CCB 总线上的 MPC85XX 芯片的设备与主存储器进行 DMA 数据传送时将无视 Cache 的存在。因此如果 SDMA 不支持与 Cache 的一致性,处理器在进行 SDMA 读操作前,必须对 Cache 进行刷新操作以保证 Cache 与存储器的一致性。当使用 SDMA 对主存储器进行写操作后,由于 SDMA 只更新了主存储器,并没有更新 Cache,也会造成主存储器和 Cache 的不一致,因此在使用 SDMA 写操作后,必须进行 Cache 无效操作,将与 SDMA 使用的内存空间相对应的 Cache 使无效后,E500 内核才能对这些数据进行读操作。

第 4 章　基于 E500 内核的 PowerPC 处理器

Freescale PQIII 系列的芯片都是基于 E500 内核的处理器。这些处理器包括 MPC8540、MPC8560、MPC8541、MPC8547 和 MPC8548 等。这一类芯片主要适用于通信领域的高端应用，此外在工业控制领域，PQIII 芯片也得到了广泛的应用。

Freescale 在近期还会推出一系列基于 E500 内核的处理器，包括一些基于 E500 内核的多核处理器。基于 E500 内核的处理器是 Freescale 高端处理器发展的主要方向，这也是本书以 E500 内核为例，介绍 PowerPC 处理器的主要原因。

4.1　基于 E500 内核的处理器

Freescale 基于 E500 V1 内核的芯片主要包括 MPC8540、MPC8560、MPC8541 与 MPC8555，基于 E500 V2 内核的芯片主要有 MPC8548，MPC8547 等。

在有些 MPC85XX 芯片内部集成了两个处理模块，一个高性能 PowerPC E500 内核和一个通信处理模块(Communications Processor Module，CPM)。此外，这类芯片系列中还提供了片内 L2 Cache、DDR 控制器、可编程中断控制器、通用 I/O 口、DMA 和 I2C 等多种接口控制器。MPC85XX 芯片结构如图 4-1 所示。

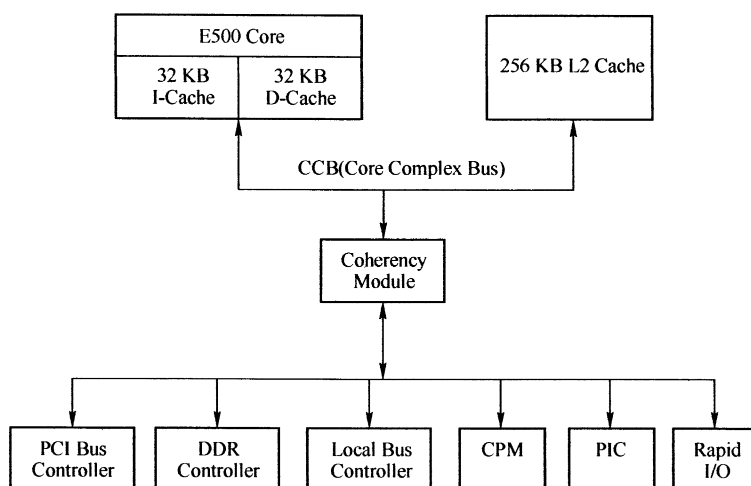


图 4-1　基于 E500 内核的 PowerPC 处理器

由上图所示，一个基于 E500 内核的 PowerPC 处理器除了 E500 内核和 L2 Cache 外，一般还包含以下模块：

- PCI 总线控制器，可以支持 PCI32、PCI64 总线。MPC8560 和 MPC8540 处理器还支持 PCI-X 总线。在 MPC8548 种还支持 PCI Express 总线。
- DDR 控制器，可以支持 64 位的 DDR 总线宽度。目前一部分 Freescale 的 MPC85XX 系

列芯片还支持 DDR-2 总线。

- Local Bus 控制器,可以与 Flash、SDRAM 及其他外部设备进行相连。
- PIC 控制器用于管理所有外部中断。MPC85XX 处理器的 PIC 控制器与 OpenPIC 构架的中断控制器兼容。
- Rapid I/O 控制器,目前有的 MPC85XX 处理器支持并行 Rapid I/O 接口,有些支持串行 Rapid I/O 接口。
- CPM,是 Freescale 用作处理网络协议的处理器,其中集成了许多与通信相关的各种接口如 TDM、ATM、同步及异步串行口等各种外设。尽管 CPM 的接口丰富,功能强大,但从计算机体系结构的角度上看并没有什么新颖之处。从处理器的设计角度上看,采用 CPM 的优点主要是利用软件采取了一种相对廉价的方法实现了与现代通信中常用的接口进行连接。

本文将简要介绍 CPM 与 PowerPC 处理器的通信机制,同时对处理器内部内存映射的寄存器、L2 Cache 与处理器体系结构有关的模块进行简单描述。有关 PIC 中断控制器的内容将结合 Linux PowerPC 的源代码进行详细说明。

4.1.1 PowerQUICC III 处理器的 CPM

1993 年,摩托罗拉半导体事业部(飞思卡尔半导体的前身)在充分理解通信系统应用的基础上,生产了第一颗包含 QUICC(Quad Integrated Communications Controller)的芯片——MC68360。QUICC 实际上是一个独立的处理器,其中的 Quad 一词源自 MC68360 中有四个串行通信控制器(SCC),实际上这颗芯片还有两个串行管理控制器(SMC)和一个串行外围接口电路(SPI)。

这颗基于 68 K 内核的芯片可以支持当时通信系统设计中需要的 10M 以太网,7 号信令系统,HDLC/SDLC,PPP(Point to Point Protocol),V.14,X.21,以及各种串行通路。在当时,这颗芯片的推出可以用横空出世来形容。

1994 年,摩托罗拉半导体事业部的工程师开始将 MC68360 中 QUICC 通信处理器模块与 PowerPC 内核相结合。摩托罗拉半导体事业部将集成了 QUICC 的 PowerPC 处理器统称为 PowerQUICC 系列,并将 QUICC 处理模块称为 CPM。

一年后,具有 PowerPC 603 内核的 MPC860 芯片面世。这颗芯片的诞生标志了一个通信处理器时代的开始,从计算机体系结构的角度上看,这颗芯片实现了将作为控制中心 PowerPC 处理器和作为数据处理中心的 CPM 分离,采用了控制通路和数据通路分开的思想。在此后的许多年间,这颗芯片在通信处理器领域中没有对手。即使是现在,这颗芯片仍然有着广泛的应用。

随后 PowerPC 内核和 CPM 处理单元的主频不断地提高,新的通信协议不断加到 CPM 中。PowerQUICC 系列芯片也从最初的 PowerQUICC I 升级到 PowerQUICC II, PowerQUICC III 和 PowerQUICC II Pro 系列。

其中 PowerQUICC II Pro 系列的某些芯片将 CPM 模块进一步升级为 QUICC Engine。QUICC Engine 技术在 CPM 技术的基础上又向前演进了一步。它可以采用两个 RISC 引擎,能够提供 4 倍于 CPM 的数据吞吐量。QUICC 引擎技术还能对 L3/L4 协议进行处理,并提供集成的多协议处理和互通功能,速度高达 1.2 Gbit/s。这些技术使 PowerPC 内核能够进一步

集中于控制通路的处理,从而提高了 PowerQUICC 处理器的整体性能。

即便如此,PowerQUICC 系列的根本设计思想并没有质的改变,CPM 与 PowerPC 内核间的通信机制也没有质的变化。

1. MPC85XX CPM 的组成

MPC85XX 是属于 PowerQUICC III 的通信处理芯片。MPC85XX 在内部集成了两个处理模块,一个模块是 PowerPC E500 处理器内核,另一个是通信处理模块(CPM)。一个典型的 CPM 一般包括以下几个单元,如图 4-2 所示。

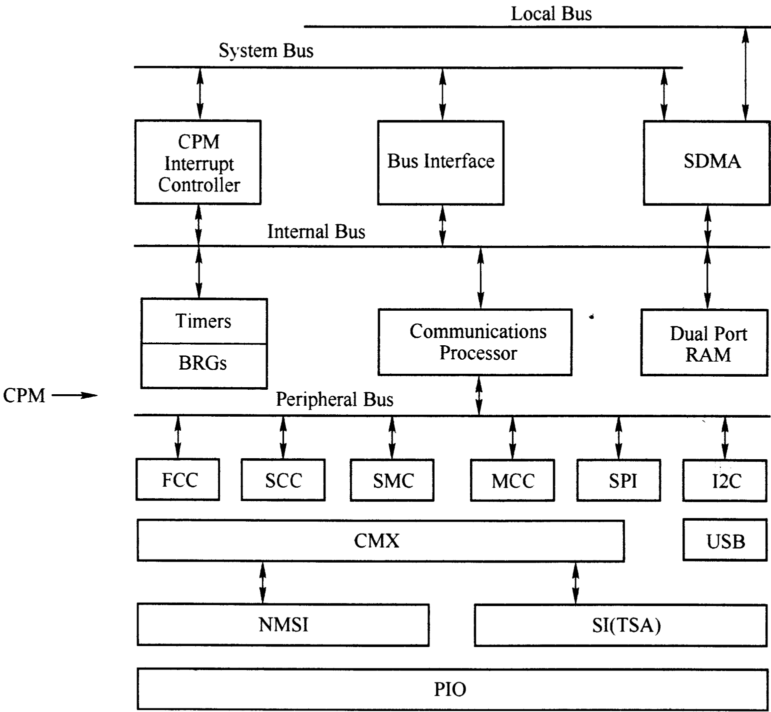


图 4-2　CPM 组成结构图

(1) 通信处理器(Communication Processor,CP)。CPM 内部集成了一颗 32 位的处理器用来管理 CPM 中的设备及中断。CP 将 CPM 中的外设与 PowerPC 隔离,并集中对设备的控制通路进行管理。这种外设管理方式可以提高 PowerPC 的访问外部设备的效率,但或多或少地增加了 PowerPC 访问外设的延时。

(2) Serial DMA。CPM 可以通过 SDMA 与 PowerPC 的系统总线(System Bus)和局部总线(Local Bus)进行数据交换。从而实现了 PowerQUICC 芯片中控制通路与数据通路的分离。

(3) 功能部件 FCC、SCC、SMC、MCC、SPI、I2C 和 USB 用来支持各种外设,如以太网、UTOPIA、TDM、UART 和 BISYNC 等。

(4) 双端口存储器(Dual Port RAM,DPRAM)。在 DPRAM 中可以存放 CP 和 SDMA 中的一些参数、CP 的可执行代码、FCC、SCC 等功能部件使用的一些临时数据。DPRAM 可以被 PowerPC 内核、CP、SDMA 访问。CP 的可执行代码主要是放在片内 ROM 中的。DPRAM 中的可执行代码主要是更新已有代码的 patch 或加入新的协议。

DPRAM 中还存放了一组重要的参数,PowerQUICC 将这些参数存放的位置称为参数

RAM(Parameter RAM)。在参数 RAM 中存放了对 CPM 中的外部设备如 FCC、SCC 的描述和定义，参数 RAM 中共存放了两类参数 RX 参数和 TX 参数。

参数 RAM 可以被 PowerPC 内核直接访问。其中 PowerPC 内核可以在处理器的运行时刻对 RX 参数和 TX 参数进行修改。在 CPM 相应外设的接收禁止后，PowerPC 处理器可以对 RX 参数进行修改；在 CPM 相应外设的发送禁止后，PowerPC 处理器可以对 TX 参数进行修改。

当 PowerQUICC 处理器需要对 CPM 相应的外设，如 SCC 的工作状态进行修改时，需要首先通过 CP 将 SCC 的接收部件或发送部件禁止，然后修改相应 SCC 的参数 RAM，最后通过 CP 重新启动 SCC 的接收或发送部件。这里要提醒读者：PowerPC 处理器在修改参数 RAM 时有些限制，有些操作必须在 CPM 禁止对应的收发后才能进行。

(5) 数据缓冲描述符(Buffer Descriptor,BD)。在 PowerQUICC 体系中，程序员可以根据需要将 BD 存放在 CPM 未用的 DPRAM 中或者主存储器中。BD 中定义了 CPM 与系统总线及其 MPC85XX 芯片局部总线使用 SDMA 进行数据传递的规则。其中 BD 中的数值由 PowerPC 内核进行填写并由 CPM 进行状态更新，CPM 可以根据 BD 中的数据自动发起 SDMA 操作将 CPM 外设中的数据根据需要传递到 DDR 或者局部总线的内存中。

驱动程序设计人员可以根据需要使用一个 BD 或者一组 BD 进行数据转送。一般都要求至少两个 BD，否则容易产生 underrun 或者 overrun。BD 共分为接收 BD 和发送 BD 两类。接收 BD 和发送 BD 都由 4 个字节组成。其中 BD 的第 0,1 字节用来存放 BD 的状态和控制信息。这些状态和控制位根据 CPM 所支持的外设的不同而有所不同；第 2,3 字节用来存放当前 BD 所能传送或接收的数据大小；第 4,5,6,7 字节用来存放数据发送或接收的地址。

(6) 波特率发生器(Baud-Rate Generator,BRG)。BRG 在 CPM 内部，可以为 CPM 内部的 SCC,FCC 等功能单元提供时钟源，也可以输出到 CPU 外部。BRG 的输入时钟可以来自 BRGCLK(BRGCLK 可以由系统时钟分频产生)，也可以来自外部输入引脚。这些时钟在 BRG 中可以根据实际需要进行分频。BRG 的工作原理如图 4-3 所示。

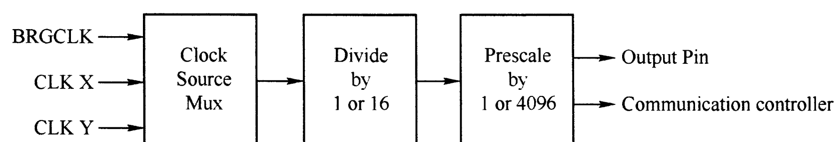


图 4-3　BRG 的工作原理

(7) CMX(CPM Multiplexing Logic)。CMX 用来将 CPM 中的各个功能部件如 FCC、SCC 与物理层设备进行连接。这些功能部件与物理层设备进行连接时共有两种模式进行选择，NMSI(Nonmultiplexed Serial Interface)或 SI(Serial Interface)。其中 NMSI 模式用来连接以太网、UTOPIA、同/异步串口等其他外部接口，而 SI 模式用来连接 TDM 设备。

(8) 并行输入/输出接口(Parallel I/O,PIO)。在 CPM 内部支持多组 PIO 接口，这些 PIO 接口可以用作通用 I/O 接口也可以根据寄存器设置作为某些设备的专用接口。

2. CPM 的工作原理

CPM 中除了具有一个 32 位的处理器以外，还有固化在 ROM 中的程序，串行通信控制器，以及与各个通信控制器相关的数据结构。

CPM 的工作原理较为复杂，理解 CPM 各个功能模块的工作原理的过程也是理解各个功能模块所支持协议的过程。操作系统一般将 CPM 各个功能模块以设备驱动程序的方式实现，因此 CPM 虽然在 PowerQUICC 处理器中占有绝对的重要地位，但在 PowerPC 处理器的体系结构中并不重要。本书仅以 SCC UART 为例简要说明 CPM 的工作原理。UART 的初始化流程如下所示：

(1) 初始化 PIO，将 PIO 的相应引脚设置为与 UART 相关的引脚。

(2) 初始化 BRG，为 UART 设定时钟频率。

(3) 初始化 CPM 的 CMX，选择 NMSI 模式。

(4) 初始化 GSMR _ H 和 GSMR _ L 寄存器。GSMR(General SCC mode Register)用来设置 SCC 的工作模式及工作参数，将 SCC 设置为 UART。

(5) 初始化 PSMR 寄存器，以确定 UART 的工作协议。

(6) 初始化 SCC1(选定 SCC1 处理 UART 接口)的参数 RAM，RxBD 和 TxBD。

(7) 通过操作 CPCR 寄存器向 CP 提交一系列命令使得参数 RAM 中的值生效。

(8) 清除 SCCE 中残留的 UART 事件。

(9) 使能中断。用户可以根据需要屏蔽或者使能中断。

(10) 在 GSMR _ L 寄存器中使能 UART 的发送和接收。

初始化完成后，驱动程序设计人员只需要操作相应的 RxBD 和 TxBD 就可以完成 UART 接口的发送和接收操作。CPM 的其他功能单元的工作原理与 UART 接口有相似之处，不过随着功能模块所支持协议的不同，初始化的过程不尽相同。CPM 功能模块所支持的协议越复杂，相应的初始化流程也越繁琐。对此，本书不再一一叙述。

4.1.2 PowerQUICC III 处理器中存储器映射的寄存器

MPC85XX 处理器中有两类寄存器，一类寄存器是 PowerPC E500 内核的内部寄存器如 MSR，HID 寄存器等，使用 mtspr，mfspr 指令可以对这些寄存器进行读写；而另一类寄存器是 MPC85XX 处理器的内部寄存器，这些寄存器采用存储器映射进行寻址，其地址空间大小为 1MB，如 LAWBAR0，LAWBAR1 寄存器等。用户可以使用 stw 和 lwz 等存储器读写指令对这些存储器映射的寄存器进行访问。MPC85XX 处理器内储器映射的寄存器根据功能可以分为以下几类。

(1) Local Access Register(局部访问寄存器)。局部访问寄存器主要用于定义 MPC85XX 的数据访问空间。包括 CCSRBAR 寄存器，LAWBAR0～7 和 LAWAR0～7 寄存器。

其中 CCSRBAR 寄存器用来确定 MPC85XX 中采用存储器映射的寄存器的基地址。此寄存器只有第 12～23 位有效，默认值为 0x000FF700，表示 MPC85XX 的内部寄存器映射在 0xFF700000 开始的 1MB 空间里。此寄存器可写，用户可以将 MPC85XX 内部寄存器的基地址映射到其他 1MB 对界的空间里。对此寄存器进行读写时需要注意同步问题。

LAWBAR0～7，LAWAR0～7 寄存器。LAWBAR，LAWAR 寄存器用来描述 MPC85XX 处理器物理地址空间的划分，此寄存器共分为 8 组，可以将 MPC85XX 处理器内 4GB 的空间划分为 8 组。其中 LAWBAR 用来指定基地址。LAWAR 用来指定此空间是用作 PCI，Local Bus，RapidIO 还是 DDR 空间。

(2) PIC Register(中断控制寄存器)。用于配置 MPC85XX 的可编程序控制器。

(3) DDR Memory Controller Memory Map(DDR 总线控制寄存器)。用于配置 MPC85XX 处理器的 DDR 总线控制器。DDR 总线控制寄存器必须根据当前处理器系统中采用的 DDR 颗粒进行配置。

(4) Local Bus Controller Register(局部总线控制寄存器)。用于配置 MPC85XX 处理器的局部总线。MPC85XX 处理器的局部总线可以与异步存储器如 NOR-FLASH 进行连接,也可以连接 SDRAM。MPC85XX 处理器的局部总线可以为用户提供 8 条片选信号,用户可以使用局部总线连接自定义的设备,与此对应,MPC85XX 处理器芯片中提供了 8 组寄存器 BR0～7 与 OR0～7,用来描述对应片选信号所访问空间的大小与属性。

(5) PCI Register(PCI 总线控制寄存器)。MPC85XX 处理器支持 PCI 2.2 规范,可以被配置成为 PCI 总线的 Host 设备或者 Agent 设备。MPC85XX 处理器的 PCI 寄存器中有许多个寄存器,其中有关 Outbound Window 和 Inbound Window 的两组寄存器值得关注。

当 MPC85XX 处理器工作在 PCI Host 模式时,Outbound Window 寄存器组用来配置当前处理器系统中的 PCI Agent 设备的物理地址空间的基地址,工作模式和地址空间大小。在 MPC85XX 处理器中有 4 组 Outbound Window 寄存器,在实际系统设计中,用户至少要使用两组空间分别对设备的 Memory 空间和 I/O 空间进行映射。

当 MPC85XX 处理器工作在 Host 模式时,Inbound Window 寄存器用来设置系统中 PCI Agent 设备的可以访问的物理地址空间。

(6) DMA Register(DMA 控制寄存器)。MPC85XX 处理器提供了 4 路 DMA 控制器可以由用户使用。E500 内核和外部设备都可以启动这类 DMA 传送,E500 内核可以通过填写寄存器的方式启动这类 DMA 传送,而外部设备需要对 MPC85XX 的外部信号 DMA_DREQ#, DMA_DACK# 与 DMA_DDONE# 进行控制以启动这类 DMA 传送。

(7) TSEC Control and Status Register(三速以太网控制状态寄存器)。MPC85XX 处理器中可以支持多个 TSEC 设备。MPC85XX 的 TSEC 的编程模式与 CPM 中的功能部件类似。对 TSEC 设备的访问也需要通过操作 BD 完成。

需要注意的是,TSEC 控制器的 RxBD 和 TxBD 不能放入 CPM 的 DPRAM 中而是放入 DDR 中。这样设计的目的一是因为 TSEC 控制器访问 CPM 内部的 DPRAM 的速度并不比访问 DDR 的速度快;二是因为 CPM 的 DPRAM 是由 E500 和 CPM 进行访问,如果 TSEC 也对 CPM 中的 DPRAM 进行访问,当网络流量过大时将会对 E500 访问 CPM 造成较大的影响,从而引起一些不可预料的错误。

(8) 其他寄存器,略。

4.1.3 L2 Cache

在 E500 内核中不包含 L2 Cache。因此 L2 Cache 的实现可以根据处理器的特点进行设置。MPC8540/8560/8541/8555 的 L2 Cache 大小为 256 KB,采用 8 路组相连方式。在这些处理器中,256 KB 的 L2 Cache 可以完全模拟成为独立的 SRAM,由处理器进行访问,也可以划出 128KB 作为 L2 Cache,128KB 作为 SRAM。

MPC85XX 的 L2 Cache 采用 front-side 方式。其中 front-side 总线是指芯片内部与外设进行连接的总线,在 MPC85XX 中这条总线就是 CCB 总线。采用 front-side bus 方式是指 L2 Cache 与 front-side bus 总线进行连接。

与 front-side 总线相对，在有的处理器系统中还有一种 Cache 连接方式，即 back-side 方式。使用 back-side 方式进行 L2 Cache 连接是指 L2 Cache 连接在 back-side 总线上，其中 back-side 总线是处理器内核与 Cache 进行连接的专用总线。

back-side 总线的频率远比 front-side 总线的频率高，因此挂接在 back-side 总线的 Cache 访问速度快。但是采用这种方式的 Cache 在进行共享一致尤其是与外设 DMA 操作进行存储一致性时，在设计上不易实现，而且所消耗的资源相对较大。一般来说，重视运算性能的处理器中在连接 Cache 时采用了 back-side 方式，而在一些相对较为简单的处理器中采用了 front-side 方式。

MPC85XX 处理器的 L2 Cache 连接在 CCB 总线，其他外设的控制器如 DDR 总线控制器，PCI 总线控制器，TSEC，CPM 也连接在这条总线上。这种连接方式比较容易实现系统的 Cache 一致性。采用这种方式的 Cache 一般采用通写方式而不是回写方式进行 Cache 更新。

因为 L2 cache 与 DDR 控制器都在 CCB 总线中，如果采用回写方式进行 Cache 更新，当进行数据写操作时，首先需要通过 CCB 总线对 Cache 进行写操作并将 Cache 行状态设为 modified，之后当更新此 Cache 行时，还要通过 CCB 总线将更新的数据回写到 DDR 中。因此采用 front-side 方式的 L2 Cache 采用通写方式进行 Cache 更新反而可以提高 CCB 总线的利用率。

MPC85XX 中 L2 Cache 的状态也相对简单，一共有 Valid、Locked 和 Stale 三种状态，分别表示当前 Cache 行是否有效、锁定、被改写。对 L2 Cache 的更新也使用 PLRU 算法。

4.2 基于 E500 内核的多处理器

本章所介绍的多处理器是指在前端总线(Core Complex Bus，CCB)挂接多个 E500 内核的多处理器。目前 Freescale 尚没有超过 2 个 E500 内核的处理器。但是在不远的将来，多核处理器，如 4 核、8 核处理器会得到越来越多的应用。

典型的对称多处理器(Symmetric Mutliple Processor，SMP)结构如图 4-4 所示。在这个系统中四个 E500 内核通过前端总线 CCB 连接。其中每个 E500 内核具有独立的 L1 Cache 和 L2 Cache，显然这种结构并非多核处理器的最优结构，但本书的目的仅是利用此结构对多核系统进行简要说明。

如图 4-4 所示，此 SMP 处理器的所有的外设挂接在前端总线 CCB 上，由四个 E500 内核共享，每一个 E500 内核共享一个 DDR 控制器，因此多个处理器内核共享同一个主存储器。这使得基于 SMP 处理器的操作系统可以相对容易、高速地在各个处理器之间传递信息。此外 SMP 处理器还共享一个中断控制器 PIC，多个处理器之间还可以通过处理器间的中断进行信息交换。

SMP 结构的处理器由于编程、使用和管理的简便性，得到了广泛的应用。但是由于系统总线带宽的限制，SMP 结构不能支持太多的处理器。太多的处理器共享系统总线时，将会降低系统的效率。

在一个典型的多机系统，如 NUMA(Non Uniform Memory Access)结构的大规模并行处理器(Massive Parallel Processor，MPP)中，SMP 结构的处理器通常会作为一个基本处理器单元(Processor Element，PE)组成一个多机系统。

在多机系统中，有两个永恒的话题：一个是多处理器间通信的延时与带宽，另一个是系统的可编程性。MPP 处理器为提高通信的带宽减少通信延时，使用了惊人的代价。而提高通信

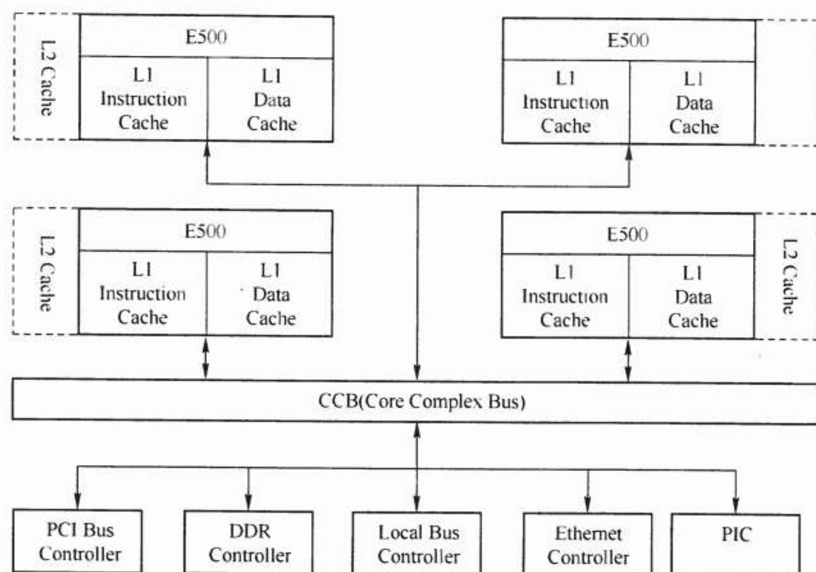


图 4-4 基于 E500 内核的对称多处理器结构图

延时的第一步就是实现 SMP 间内存访问的同步和 Cache 的共享一致性。E500 内核中在指令和系统总线级对这些同步和一致性进行了必要的支持。

4.2.1 SMP 的同步机制

SMP 的同步包括数据同步、存储管理同步、原子操作和锁机制。数据同步主要包括 SMP 的 Cache 一致性协议和对共享数据的访问。

在 E500 内核中，当 HID1 寄存器中 ABE 位为 1 时将使能地址广播操作。此时 dcbst, dcblc, icblc, icblc, dcbf, mbar, msync, tlbsync, tlbivax, icbi 指令将在 CCB 总线上进行广播，将本处理器内核中发生的事件通知给其他在 CCB 总线上的处理器内核。

SMP 一般使用原子操作或者自旋锁对共享数据进行访问。E500 内核支持两条用于原子操作和锁操作的指令，分别是 lwarx 和 stwcx. 指令。这两条指令成对出现，用来支持原子操作和锁操作。为支持这些操作，E500 内核的 CCB 总线中包含一个 RESERVE 状态位和 RESERVE 地址。在多处理器系统中，这个 RESERVE 位和 RESERVER 地址可以通过 CCB 总线进行同步并保证其一致性。CCB 总线还支持存储器原子访问周期，如 read atomic 和 RWITM atomic 总线周期。

1. lwarx 和 stwcx. 指令

lwarx 和 stwcx. 指令主要用来实现原子操作和锁操作。

lwarx 指令的格式为"lwarx rD, rA, rB"，其指令描述如下所示：

```
if rA = 0 then a ← 0
  else a ← (rA)
EA ← a + rB
RESERVE ← 1
RESERVE_ADDR ← read_addr(EA)
rD ← (32)0 || MEM(EA, 4)
```

lwarx 指令的执行流程如下所示：

(1) 首先需要计算当前 lwarx 指令使用的有效地址 EA，该有效地址为 rA + rB(或者 rB)。

(2) 然后把 E500 内核中的 RESERVE 位置 1，并将该指令访问的有效地址 EA 对应的物理地址放入 RESERVER _ ADDR 中，将 rA + rB(或者 rB)数据放入 rD 中。

lwarx 指令对通用寄存器 rD 的影响与 lwz 指令完全相同，只是该指令会将 E500 内核中的 RESERVE 位和 RESERVER _ ADDR 寄存器进行设置。该指令的英文含义为 Load Word and Reserve Indexed。其中的"Reserve Indexed"就是指对 RESERVE 和 RESERVER _ ADDR 进行设置。

stwcx. 指令的格式为"stwcx. rS,rA,rB"，其指令描述如下所示：

```
if rA = 0 then a ← 0
  else a ← (rA)
EA ← a + rB
if RESERVE then
  if RESERVE _ ADDR = read _ addr(EA) then
    MEM(EA, 4) ← rS[32:63]
    CR0 ← 0b00 || 0b1 || XERso
  else
    u ← undefined 1 - bit value
    if u then MEM(EA, 4) ← rS[32:63]
    CR0 ← 0b00 || 0bu || XERso
  RESERVER ← 0
else
  CR0 ← 0b00 || 0b0 || XERso
```

由上可知 stwcx. 指令的执行结果与 lwarx 指令结果的执行有关。

(1) stwcx. 指令首先计算当前 stwcx. 指令使用的有效地址 EA，该有效地址为 rA + rB(或者 rB)。

(2) 然后该指令对 RESERVE 位进行检查，如果 RESERVE 位不为 1，即在该指令之前程序没有执行过 lwarx 指令，该指令将 CR 寄存器的 CR0 字段的 EQ 位清零。此时该指令将不对有效地址 EA 进行任何操作。

(3) 如果 RESERVE 位为 1，stwcx. 指令将对 RESERVE _ ADDR 寄存器进行进一步检查。如果在 RESERVE _ ADDR 寄存器与有效地址 EA 对应的物理地址相同，则将存放在 rS 寄存器中的数据写入 EA 中；如果在 RESERVE _ ADDR 寄存器与有效地址 EA 对应的物理地址不相同，则不将 rS 中的数据写入 EA 中。

(4) 最后该指令将 RESERVE 位清零。

系统软件可以使用 lwarx 和 stwcx. 指令构建原子操作和 SMP 中常用的自旋锁。

2. 原子操作

Linux PowerPC 可以使用 lwarx 和 stwcx. 指令实现某些原子操作。

首先 lwarx 指令将要进行更新的数据读出，同时将 RESERVE 位和 RESERVE _ ADDR 寄存器初始化，再将更新的数据进行各种运算操作，最后使用 stwcx. 指令将运算结果写回。

如果 stwcx. 指令执行成功，则 EQ 位为 1，表示在 lwarx 指令和 stwcx. 指令之间的操作并没有被打破，本次原子操作成功；否则将重新使用 lwarx 指令对 RESERVE 位和 RESERVE _ ADR 寄存器进行初始化，然后对该数据继续更新，直到 stwcx. 指令成功写入数据。E500 内核中原子操作并不是一条指令而是通过查询来实现。常用的原子操作有 Fetch and add/store，Test and set 等。

(1) Fetch and Store 原子操作

```
loop: lwarx r5, 0, r3
      stwcx. r4, 0, r3
      bne loop
```

首先将存放在寄存器 r5 中的数据保存到寄存器 r3(该寄存器用来存放临界地址)，同时将 RESERVE 位置 1 并将 RESERVE _ ADR 寄存器置为寄存器 r3 中的数值。

然后使用 stwcx. 指令将寄存器 r4 的数值存放到临界地址中。如果 stwcx. 指令没有将数据存放到临界地址中，CR 寄存器 CR0 字段的 EQ 位为 0，此时需要重新使用 lwarx 指令对 RESERVE 位和 RESERVE _ ADR 寄存器进行重置，然后再对此临界区域进行操作，直到成功地将数据写入临界地址中。当 EQ 位为 1，表示 stwcx. 指令成功的将数据写入内存中，并完成 Fetch and Store 原子操作。

对于多内核处理器，有可能多个内核都执行 Fetch and Store 原子操作，如果其他内核执行 stwcx. 指令有可能会改变 RESERVE 位和 RESERVE _ ADDR，从而导致 stwcx. 指令不会成功地将数据写入到指定的临界区域中。

当两个处理器 P0 和 P1 同时进行 Fetch and Store 原子操作时，有可能出现一个极端情况，如图 4-5 所示。

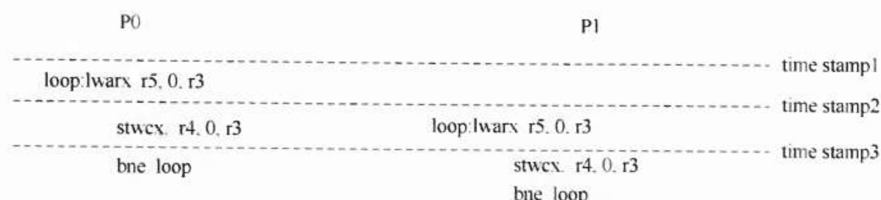


图 4-5 Fetch and Store 原子操作

1) 在 time stamp1 时刻，P0 内核执行 lwarx 指令将 E500 内核的 RESERVE 位置 1，同时将 RESERVE _ ADDR 寄存器置为 r3 寄存器中存放的物理地址。

2) 在 time stamp1 时刻和 time stamp2 时刻之间，P2 内核也开始执行 lwarx 指令将 E500 内核的 RESERVE 位置 1，同时将 RESERVE _ ADDR 寄存器置为 r3 寄存器中存放的物理地址。为简化问题，我们假定 P0 和 P1 内核中存放的 r3 寄存器内容相同。

3) 在 time stamp2 时刻，P0 内核执行 stwcx. 指令将数据写入到指定的临界区域中，同时将 RESERVE 位置 0 和 CR 寄存器 CR0 字段的 EQ 位置 1。

4) 在 time stamp2 时刻和 time stamp3 时刻之间，P1 内核也将执行 stwcx. 指令。但是此时 RESERVE 位已经为 0，因此该指令不能将数据写入到指定的临界区域中，此时 CR 寄存器 CR0 字段的 EQ 位将被置 0。

5) 在 time stamp3 时刻，P0 内核将执行 bne loop 指令，此时因为 P0 内核的 EQ 为 1，该转

移指令将不会做转移操作，P0 内核将完成 Fetch and Store 原子操作。

6) 在 time stamp3 时刻之后，P1 内核也将执行 bne loop 指令，此时因为 P1 内核的 EQ 为 0，bne loop 指令将转移到 loop 标签处，继续执行 lwarx 指令，然后再使用 stwcx. 指令将数据写入到临界区域中。

E500 内核就是使用这类指令来完成原子操作。实际上所有支持 SMP 结构的处理器内核都会支持这一类指令。在 MIPS 处理器中，也有一对指令分别是 ll 和 sc 指令完成类似的原子操作。

(2) Fetch and Add 原子操作

```
loop: lwarx r5, 0, r3
      add r0, r4, r5
      stwcx. r0, 0, r3
      bne loop
```

Fetch and Add 操作与 Fetch and Stroe 原子操作类似，只是在 lwarx 和 stwcx. 指令之间加入了一条 add 指令。事实上，可以在这两条指令之间加入一些其他的指令，如与，或等等其他指令，以生成 Fetch and And，Fetch and Or 等等操作。

使用 lwarx 和 stwcx. 实现原子操作需要将程序放入 lwarx 和 stwcx. 指令之间，并将结果通过 stwcx. 指令存入。采用这种方法可以实现一些简单的原子操作，这些原子操作一次只能通过 stwcx. 指令对一个单个临界数据进行原子操作。对于一个较大的临界区进行访问时，系统软件需要使用锁操作完成。

3. 自旋锁

在多处理器内核中运行的操作系统，如 SMP 结构的处理器，需要使用自旋锁访问临界区域。在 Linux 系统中，当使能 preemptible 功能时，运行在单处理器内核中的操作系统也需要使用自旋锁对临界数据区域进行保护。

当 Linux 系统使能 preemptible 功能时，进程在 Linux 内核中执行时，调度程序可以将当前进程切换到其他进程中运行，此时其他进程可能会对临界区域进行操作，从而有可能破坏存放在临界区域中的数据。因此如果系统程序员对 Linux 内核进行修改开发，需要保证代码可以支持 preemptible 功能时，这些代码必须需要使用自旋锁将临界数据进行保护。

Linux 系统使用 spin_lock 类函数实现自旋锁，在 ./include/linux/spinlock.h 文件中包含了一系列宏定义，如 spin_lock，spin_lock_irqsave，spin_lock_irq 实现 Linux 系统的自旋锁，spin_lock 类函数的实现与 preemptible 功能紧密相连。

在 Linux PowerPC 中，spin_lock 函数最终调用 ./include/asm-powerpc 目录的 spinlock.h 文件中的__spin_trylock 函数，在这个函数中使用了 lwcrx 和 stwcx. 指令实现自旋锁的功能。Linux 系统对临界区域的访问如图 4-6 所示。

多处理器系统对临界区域进行访问时，必须使用自旋锁对临界数据进行访问。在单处理器系统中，如果操作系统支持 preemptible，则对临界数据的访问也需要使用自旋锁。注意，在 Linux 系统中，如果一个进程使用了自旋锁对临界数据进行保护后，该进程不能被其他进程抢占。

自旋锁实际上是一个被 lwcrx 和 stwcx. 指令保护起来的数据。自旋锁为 1 时表示已对临界数据段加锁，为 0 时表示临界数据段没有加锁。

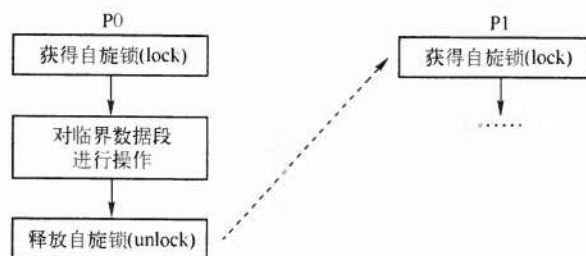


图 4-6　使用自旋锁对临界区域进行访问

假设系统中使用 lock 和 unlock 函数实现对自旋锁的加锁和解锁操作。其中 lock(int lock)函数将 lock 参数置 1，而 unlock(int lock)函数将 lock 参数置 0。自旋锁 Lock 和 unlock 的汇编实现如下所示。其中寄存器 r3 中保存 lock 函数和 unlock 函数的输入参数 lock。

```
lock:
  li r0, 1
lock_loop:
  lwarx r4, 0, r3
cmpwi r4, 0
  bne lock_loop
  stwcw. r0, 0, r3
  bne lock_loop
  isync
```

以上程序的详解如下：

(1) 这段程序首先将数据 1 保存在寄存器 r0 中，然后使用 lwarx 指令将存放在寄存器 r3 中的 lock 参数暂时存放在寄存器 r4 中。

(2) 对存放在寄存器 r4 中的数据进行判断，如果该值为 0，表示 lock 参数没有被其他处理器中运行的进程使用，bne 指令将不进行跳转。如果该值不为 0，则表示其他处理器正在使用该自旋锁，因此当前进程必须对存放在寄存器 r4 中的值反复查询，直到该值为 0 后，即其他处理器释放了该自旋锁后才能够继续执行。

(3) 使用 stwcwx. 指令试图将数值"1"存放在寄存器 r3 中，即将该自旋锁加锁。

(4) 如果没有加锁成功，则跳转到 lock_loop 标签处，继续对自旋锁进行加锁，否则执行 isync 指令进行指令同步，完成整个加锁过程。

unlock 函数的汇编实现如下。

```
unlock: msync
        li r0, 0
        stw r0, 0(r3)
```

解锁的过程较为简单，只需要在存储器同步指令后，简单地将自旋锁置为 0 即可。

处理器使用自旋锁对临界数据段进行访问时，务必快速完成，以保证系统的效率。在图 4-6 所示的例子中，处理器 1 在没有获得自旋锁之前，需要不断地通过 lwcrx 指令从内存中读取数据，不断地将 RESERVE 位和 RESERVE 地址进行重置，这些操作不仅会占用处理器的运行时间，而且会占用 CCB 总线的带宽。

4.2.2 SMP结构处理器的Cache共享一致性

SMP结构处理器一般采用总线监听法实现Cache共享一致性。Cache共享一致性一般是指在SMP结构处理器中共享L1 Cache。因为不同类型的SMP,其L2 Cache的设计不尽相同。有时在SMP结构处理器中的多个处理器内核共享一个L2 Cache,有时每一个处理器内核都有各自的L2 Cache。

MESI协议是目前最流行的基于总线监听法的Cache共享一致性协议。与基于单处理器内核的Cache一致性相比,多处理器内核的Cache一致性的实现较为复杂。基于多处理内核的MESI协议增加了许多内容,状态转化也较为复杂。

在多处理器内核中,MESI四位的定义如下:

- M(Modified)位。M位为1时表示L1 Cache行中包含的数据无效,它既不在本处理器内核的L1 Cache中,也不在其他处理器内核的L1 Cache中。当处理器对这个L1 Cache行进行操作时,必然会导致Cache失效,从而引发系统总线的读写周期,将Cache行中数据与内存中的数据同步。
- E(Exclusive)位。E位为1时表示L1 Cache行中包含的数据有效,而且该数据仅在本处理器内核中,而在其他处理器内核的L1 Cache中没有副本。
- S(Shared)位。S位为1表示L1 Cache行中包含的数据有效,而且在本处理器内核和至少在其他处理器内核中有一个副本。
- I(Invalid)位。I位为1表示L1 Cache行中没有有效数据,该L1 Cache行没有使能。E500内核使用PLRU算法对Cache行进行替换时,将首先替换状态为Invalid的Cache行。

MESI协议定义了四种状态以保证Cache的共享一致性。分别为Invalid,Shared Unmodified,Exclusive Modified,Exclusive Unmodified状态。

- Invalid状态。Cache行处于Invalid状态表示当前Cache行的MESI中只有I位有效,此时当前L1 Cache行没有被使用。
- Shared Unmodified状态。Cache行处于这种状态时,MESI中只有S位有效,表示当前L1 Cache行中的数据被多个处理器内核共享。
- Exclusive Modified状态。Cache行处于这种状态时,MESI中E位和M位有效,表示当前L1 Cache行的数据已经被改写,但是在其他处理器内核中没有副本。
- Exclusive Unmodified状态。Cache行处于这种状态时,MESI中E位有效,表示此L1 Cache行中的数据有效。

处理器内核对Cache进行操作时,Cache行的四种状态将进行转换。本节假定用户使用回写方式管理L1 Cache。基于MESI协议的L1 Cache行状态的转换如图4-7所示。

当处理器内核访问存储器时发生读写命中(Read/Write Miss),读写失效(Read/Write Hit)或者处理器执行使无效命令(Invalidate Command)等操作时,将引发Cache行的状态转换。

为简便起见,我们假定一个处理器系统由四个内核组成,当前进行存储器访问的处理器内核称为内核A,而其他处理器内核分别被称为内核B,C和D。并一次分析发生Cache读命中、读失效、写命中和写失效情况下,Cache行状态的转换。

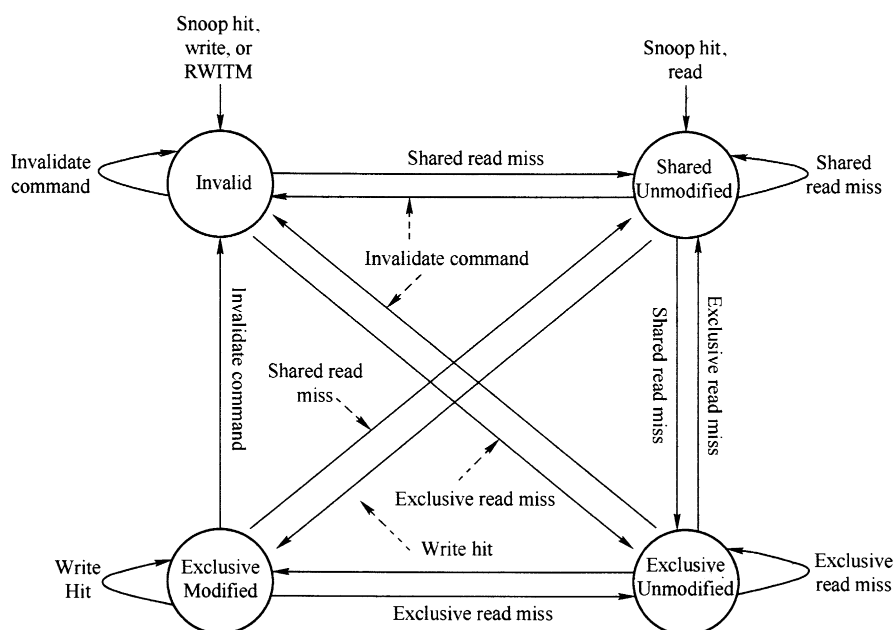


图 4-7　基于 MESI 协议的 Cache 行状态转换

1. Cache 的读命中与读失效

在多核系统中，内核 A 进行存储器读操作时，首先会访问内核 A 中的 L1 Cache。内核 A 访问 Cache 时有可能会出现两种情况：一种是访问的数据在 Cache 中命中；另一种是失效。在多核系统中，Cache 读命中的概率小于在单核系统中读命中的概率。但是这个概率也相当高。

如果内核 A 进行存储器读操作时，被访问的数据在 Cache 中命中，此时该 Cache 行的状态或者为 Exclusive Unmodified 或者为 Shared Unmodified。在这种情况下，内核 A 将从该 Cache 行中直接获得数据，而不需要启动外部总线周期，也不需要改变该 Cache 行的状态。

当内核 A 进行存储器读操作时，被访问的数据没有在某个 Cache 行中命中。此时需要根据 Cache 行状态分别进行讨论。

(1) 如果当前 Cache 行的状态为 Invalid，内核 A 将启动外部总线周期从存储体系中获得该数据，同时将更新相应的 Cache 行，并将该 Cache 行的状态更改为 Exclusive Unmodified。

(2) 如果当前 Cache 行的状态为 Exclusive Modified，此时内核 A 对读失效时的处理较为复杂。当发生读失效时，处理器不仅会改变内核 A 的 Cache 状态，也可能会影响其他内核的 Cache 状态。

Cache 行的状态为 Exclusive Modified，Cache 读失效处理需要分以下几种情况进行讨论。当发生此类读失效之后，内核 A 的 Cache 控制器将首先发出总线监听命令(snoop request)到进行 Cache 共享的内核 B，C 和 D。

(1) 如果只有内核 B 中的 Cache 中包含内核 A 所需的数据副本，而且在内核 B 中该 Cache 行的状态为 Shared Unmodified，内核 B 将通知内核 A 当前需要访问的数据已经在系统总线中“被共享”。此时内核 A 将启动外部总线周期，读取数据并更新内核 A 的 Cache 行，并将内核 A 的 Cache 行的状态更改为 Shared Unmodified。

(2) 如果只有内核 B 的 Cache 中包含内核 A 所需的数据副本，而且在内核 B 中 Cache 行

的状态为 Exclusive Unmodified，内核 B 将通知内核 A 当前需要访问的数据在系统总线中“具有备份”，同时将内核 B 的 Cache 行的状态由 Exclusive Unmodified 改变为 Shared Unmodified。之后，内核 A 将启动外部总线周期，读取数据更新 Cache 行，并将当前 Cache 行的状态更改为 Shared Unmodified。在有的多核处理器中，内核 A 可以不从内存体系中读取数据，而是直接将内核 B 中的数据备份传递给内核 A。

(3) 如果有一个内核，如内核 B 中的 Cache 中包含内核 A 所需的数据副本，但是内核 B 的 Cache 行状态为 Exclusive Modified 时，内核 B 将会中止内核 A 发起的总线监听命令，并从内存体系中读取数据以更新内核 B 的 Cache，并将内核 B 的 Cache 行状态更新为 Shared Unmodified，同时将从内存系统中获得的数据传递给内核 A，并将内核 A 相应的 Cache 行状态更改为 Shared Unmodified。

(4) 如果在当前处理器的其他内核，如内核 B，C，D 中都没有内核 A 所需的数据副本，内核 A 将从内存体系中读取数据以更新 Cache 行，并将当前 Cache 行的状态更改为 Exclusive Unmodified。

2. Cache 的写命中与写失效

与存储器读操作类似，在多核系统中，当内核 A 进行存储器写操作时，首先会访问 Cache。访问 Cache 时有可能出现两种情况：一种是访问的数据在 Cache 中命中；另一种是失效。与存储器的读操作相比，进行存储器写操作时，Cache 的处理较为复杂。

(1) 内核 A 访问存储器时，发生了 Cache 写命中，则表示内核 A 需要更新的数据在内核 A 的 Cache 行中命中。此时需要根据当前 Cache 行的状态，分以下几种情况进行讨论。

1) 如果内核 A 的 Cache 行的状态为 Shared Unmodified 时，内核 A 的 Cache 控制器将发出总线写无效命令(invalidate)到内核 B，C 和 D，将内核 B，C 和 D 中有此 Cache 行副本的 Cache 行进行写无效操作。

如果内核 B，C 和 D 的 Cache 中具有内核 A 的 Cache 副本，则其相应的 Cache 行状态为 Shared Unmodified。此时写无效命令将会把在内核 B，C 或者 D 中的 Cache 行状态更改为 Exclusive Modified，同时内核 A 将数据写入内核 A 相应的 Cache 行中，然后将内核 A 的 Cache 行状态更改为 Exclusive Modified。

2) 如果内核 A 的 Cache 行的状态为 Exclusive Unmodified，内核 A 直接将数据写入内核 A 的相应 Cache 行中，并将此 Cache 行状态更改为 Exclusive Modified，而不需要发出总线写无效命令。

3) 如果内核 A 的 Cache 行的状态为 Exclusive Modified，内核 A 直接将数据写入内核 A 的相应 Cache 行中，不需要改变此 Cache 行的状态，也不需要发出总线写无效命令。

(2) 当内核 A 访问存储器时，发生了 Cache 写失效，表示内核 A 需要更新的数据不在内核 A 的 Cache 行中。此时，内核 A 的 Cache 控制器将发出 RWITM(Read with intent to modify) 命令通知内核 B，C 和 D 进行 Cache 一致性处理。

1) 如果内核 B，C 或者 D 中的一个内核，如内核 B 的 Cache 行中包含内核 A 所需更新的数据副本，此时其 Cache 行的状态为 Exclusive Unmodified。

内核 A 的 RWITM 命令将内核 B 的 Cache 行状态更改为 Invalid。同时将内核 A 将根据回写算法，首先将内核 A 的 Cache 进行 Cache 替换，并将相应的数据写入 L1 Cache，然后将内核 A 的 Cache 行状态更改为 Exclusive Modified。

2）当其他内核B,C和D中的有一个或者多个内核的Cache中包含内核A所需更新的数据一个或者多个副本,其Cache行中的状态为Shared Unmodified。

内核A的RWITM命令将这些内核的Cache行的状态都更改为Invalid,同时根据回写算法,首先将内核A的Cache进行Cache替换,并将相应的数据写入L1 Cache,然后将内核A的Cache行状态更改为Exclusive Modified。

3）当其他内核,如内核B中的Cache包含内核A所需更新的数据副本,但是内核B的Cache行的状态为Exclusive Modified时,内核B首先需要将此Cache行的数据回写到内存中,之后再将此Cache行中的状态更改为Invalid。同时将内核A将根据回写算法,首先将内核A的Cache行进行替换,并将相应数据写入L1 Cache,同时将内核A的Cache行状态更改为Exclusive Modified。

在发生读写失效时,使用MESI协议进行Cache共享一致性的处理较为复杂。然而MESI协议仅规定了在进行Cache一致性处理时,在各个处理器内核的Cache的状态转移,在一个实际的系统中,多核的Cache一致性可能比本书描述的过程还要复杂。

有的MPP系统还采用了DASH法,即目录法结构维护全机的Cache共享一致性模式,对此感兴趣的读者,可以阅读Daniel Lenoski,James Laudon等人的论文"The DASH Prototype: Implementation and Performance"以了解DASH的详细结构,本书不对此进行描述。

4.3 大端与小端

"端模式"(Endian)源自斯威夫特写的《格列佛游记》。这本书根据将鸡蛋敲开的方法不同,把人分为两类。从圆头开始敲鸡蛋的人被称为Big Endian,从尖头开始敲鸡蛋的人被称为Littile Endian。小人国的内战就源于吃鸡蛋时是究竟从大头敲开还是从小头敲开。在计算机业Big Endian和Little Endian也几乎引起一场战争。

在计算机业界,Endian表示数据在存储器中的存放顺序。如果将一个32位的整数0x12345678存放到一个整型变量(int)中,这个整型变量采用大端和小端模式在内存中的存储位置如表4-1所示。为简单起见,本书使用OP0表示一个32位数据的最高字节(Most Significant Byte,MSB),使用OP3表示一个32位数据最低字节(Least Significant Byte,LSB)。

表4-1 数据按照大端和小端模式在内存中的存放位置

地址偏移	大端模式	小端模式
0x00	12(OP0)	78(OP3)
0x01	34(OP1)	56(OP2)
0x02	56(OP2)	34(OP1)
0x03	78(OP3)	12(OP0)

由表4-1可见,对数据进行存放时,采用大小端模式的主要区别在于存放的字节顺序。大端模式将高字节存放在低地址,小端模式将低字节存放在低地址。采用大端模式进行数据存放符合人类的正常思维,而采用小端模式进行数据存放利于计算机处理。到目前为止,是采用大端还是小端进行数据存放,孰优孰劣没有定论。

有的处理器系统采用了小端模式进行数据存放,如Intel的奔腾。有的处理器系统采用了

大端模式进行数据存放，如 IBM 半导体和 Freescale 的 PowerPC 处理器。不仅对于处理器，一些外设也存在着使用大端或者小端进行数据存放的选择。

因此在一个处理器系统中，有可能存在大端和小端模式同时存在的现象。这一现象给系统的软硬件设计带来了不小的麻烦，这要求系统设计工程师必须深入理解大端和小端模式的差别。大端与小端模式的差别体现在处理器的寄存器、指令集、系统总线等各个层次中。

4.3.1 从软件的角度理解端模式

从软件的角度看，不同端模式的处理器进行数据传递时需要考虑端模式的问题。例如进行网络数据传递时，必须考虑端模式的转换。有 Socket 接口编程经验的程序员一定使用过以下几个函数用于大小端字节序的转换。

- #define ntohs(n) //16 位数据类型网络字节顺序到主机字节顺序的转换
- #define htons(n) //16 位数据类型主机字节顺序到网络字节顺序的转换
- #define ntohl(n) //32 位数据类型网络字节顺序到主机字节顺序的转换
- #define htonl(n) //32 位数据类型主机字节顺序到网络字节顺序的转换

互联网使用的网络字节顺序采用大端模式进行编址，而主机字节顺序根据处理器的不同而不同，如 PowerPC 处理器使用大端模式，而 Pentuim 处理器使用小端模式。

大端模式处理器的字节序到网络字节序不需要转换，此时 ntohs(n) = n，ntohl(n) = n；而小端模式处理器的字节序到网络字节必须要进行转换，此时 ntohs(n) = __swab16(n)，ntohl = __swab32(n)。__swab16 与__swab32 函数定义如下所示：

```
#define __swab16(x)
{
  __u16 __x = (x);
  ((__u16)(
    (((__u16)(__x) & (__u16)0x00ffU) << 8) |
    (((__u16)(__x) & (__u16)0xff00U) >> 8) ));
}
#define __swab32(x)
{
  __u32 __x = (x);
  ((__u32)(
    (((__u32)(__x) & (__u32)0x000000ffUL) << 24) |
    (((__u32)(__x) & (__u32)0x0000ff00UL) << 8) |
    (((__u32)(__x) & (__u32)0x00ff0000UL) >> 8) |
    (((__u32)(__x) & (__u32)0xff000000UL) >> 24) ));
}
```

PowerPC 处理器提供了 lwbrx、lhbrx、stwbrx、sthbrx 四条指令处理字节序的转换以优化__swab16 和__swap32 函数。此外 PowerPC 处理器的 rlwimi 指令也可以用来实现__swab16 和__swap32 这类函数。Linux PowerPC 定义了一系列有关字节序转换的函数，其详细定义在 ./include/asm-powerpc/byteorder.h 文件中。

程序员在对普通文件进行处理也需要考虑端模式问题。在大端模式处理器下对同一文件

的 32 位、16 位读写操作所得到的结果可能与小端模式处理器不同。读者单纯从软件的角度理解远远不能真正理解大小端模式的区别。事实上，要真正理解大小端模式的区别，必须从指令集，寄存器和数据总线上的层面上深入研究。

4.3.2 从系统的角度理解端模式

除了 4.2.1 节中，软件中对不同端模式编程上的差异，硬件设计也存在由于端模式而带来的问题。从系统的角度上看，端模式问题对软件和硬件的设计带来了不同的影响。如果在一个处理器系统中，大小端模式同时存在时，设计者必须对这些不同端模式的访问进行特殊的处理。

在网络协议的软件设计中，使用小端模式的处理器需要在软件中处理端模式的转变。而多数外设都采用小端模式，包括一些在网络设备中使用的 PCI 总线、Flash 等设备，这也要求硬件工程师在硬件设计中注意端模式的转换。

本书中的小端外设是指这种外设中的寄存器以小端模式进行存储，如 PCI 设备的配置空间、NOR FLASH 中的寄存器等等。

对于有些设备，如 DDR 颗粒，没有以小端模式存储的寄存器，因此从逻辑上讲并不需要对端模式进行转换。在设计中，只需要将双方数据总线进行一一对应的互连，而不需要进行数据总线的转换。

从实际应用的角度看，采用小端模式的处理器需要在软件中处理端模式的转换，因为采用小端模式的处理器在与小端外设互连时，不需要任何转换。

而采用大端模式的处理器需要在硬件设计时处理端模式的转换。大端模式处理器需要在寄存器、指令集、数据总线与小端外设的连接等多个方面进行处理，以解决与小端外设连接时的端模式转换问题。

在寄存器和数据总线的位序定义上，大小端模式的处理器有所不同。

一个采用大端模式的 32 位处理器，如基于 E500 内核的 MPC8541，将其寄存器的最高位（msb，most significant bit）定义为 0，最低位（lsb，least significant bit）定义为 31；而小端模式的 32 位处理器，将其寄存器的最高位定义为 31，低位地址定义为 0，如图 4-8 所示。与此相对应，32 位大端模式处理器数据总线的最高位为 0，最低位为 31；32 位小端模式处理器的数据总线的最高位为 31，最低位为 0。

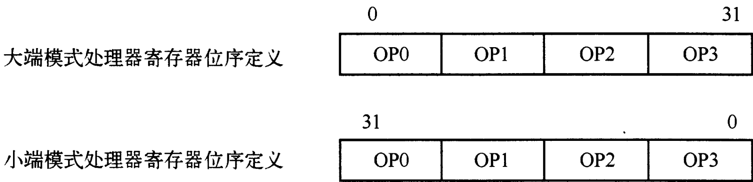


图 4-8 大小端模式处理器的寄存器的定义

大小端模式处理器外部总线的位序根据所采用的数据总线是 32 位、16 位和 8 位，大小端处理器外部总线的位序有所不同。

- 大端模式下，32 位数据总线的 msb 是第 0 位，MSB 是数据总线的第 0～7 字段；而 lsb 是第 31 位，LSB 是第 24～31 字段。小端模式下，32 位总线的 msb 是第 31 位，MSB 是数据总线的第 31～24 位，lsb 是第 0 位，LSB 是 7～0 字段。

● 大端模式下,16 位数据总线的 msb 是第 0 位,MSB 是数据总线的第 0～7 字段;而 lsb 是第 15 位,LSB 是第 8～15 字段。小端模式下,16 位总线的 msb 是第 15 位,MSB 是数据总线的第 15～7 位,lsb 是第 0 位,LSB 是第 7～0 字段。

● 大端模式下,8 位数据总线的 msb 是第 0 位,MSB 是数据总线的第 0～7 字段;而 lsb 是第 7 位,LSB 是第 0～7 字段。小端模式下,8 位总线的 msb 是第 7 位,MSB 是数据总线的第 7～0 位,lsb 是第 0 位,LSB 是 7～0 字段。

由以上分析,我们得知对于 8 位、16 位和 32 位宽度的数据总线,采用大端模式时数据总线的 msb 和 MSB 位置不会发生变化,而采用小端模式时数据总线的 lsb 和 LSB 位置也不会发生变化。

为此,大端模式的处理器对 8 位、16 位和 32 位的内存访问(包括外设的访问)一般都包含第 0～7 字段,即 MSB。小端模式的处理器对 8 位、16 位和 32 位的内存访问都包含第 7～0 位,小端方式的第 7～0 字段,即 LSB。

由于大小端处理器其 8 位、16 位和 32 位宽度的数据总线的定义不同,因此需要分别讨论这些数据总线在系统级别上如何处理端模式转换。在实际应用中,并没有大端外设,因此本书只对大端处理器访问小端外设进行说明。

1. 大端处理器访问 32 位小端外设

大端处理器采用 32 位总线访问小端外设时,其数据总线的第 0～7 位用来处理 OP0,第 8～15 位用来处理 OP1,第 16～23 位用来处理 OP2,第 24～31 位用来处理 OP3。而 32 位的小端设备使用数据总线的第 31～24 位用来处理 OP0,第 23～16 位用来处理 OP1,第 15～8 位用来处理 OP2,第 7～0 位用来处理 OP3。

PowerPC 处理器,如 MPC8541,可以使用 stw 和指令对 32 位的外部设备进行访问。在这些指令结束后,存放在外部设备的数据将被读入 MPC8541 的通用寄存器中。为保证软件的一致性,当访问结束后,存放在通用寄存器的字节序,即 OP0,OP1,OP2 和 OP3 必须和存放在小端外设的字节序一致。此时在使用大端处理器的数据总线连接小端外设时需要按照某种拓扑结构连接以保证软件的一致性。大端处理器数据总线与小端外设进行连接的拓扑结构如图 4-9 所示。

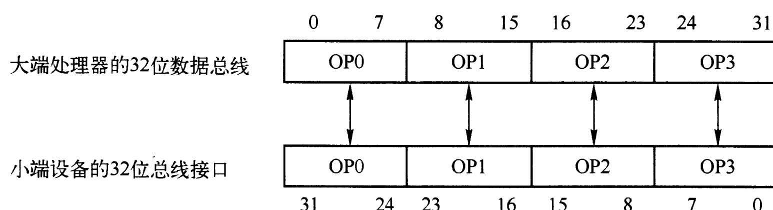


图 4-9 大端处理器与小端外设的 32 位连接

大端处理器访问小端设备时,需要将各自的 OP0～OP3 字段直接相连。在大端处理器中,32 位数据总线的最高位为 0,最低位为 31;而小端设备的最高位为 31,最低位为 0。因此硬件工程师在进行信号连接时需要将大端处理器的 0～31 位与小端设备的 31～0 位一一对应进行互连。

2. 大端处理器对 8、16 位小端外设进行访问

对于 32 位处理器,用来连接外设的总线一般是 32 位。因此体系结构设计师在进行大端

处理器总线设计时有两种选择：一是采用32位总线的高端部分(第0～15字段)连接8、16位的小端设备；二是采用低端部分(第16～31字段)。

PowerPC处理器使用32位总线的高端部分，即数据总线的第0～15位连接16位的小端设备，使用0～7位连接8位的小端设备。

PowerPC处理器采用16位总线与16位的小端外设进行访问时，PowerPC处理器的16位数据总线的第0～7位用来处理OP0，第8～15位用来处理OP1。而16位的小端设备使用数据总线的第15～8位用来处理OP0，第7～0位用来处理OP1。

PowerPC处理器采用8位总线与8位的小端外设进行访问时，PowerPC处理器的8位数据总线的第0～7字段用来处理OP0。而8位的小端设备使用数据总线的第7～0位用来处理OP1。大端处理器与小端外设的拓扑结构如图4-10所示。

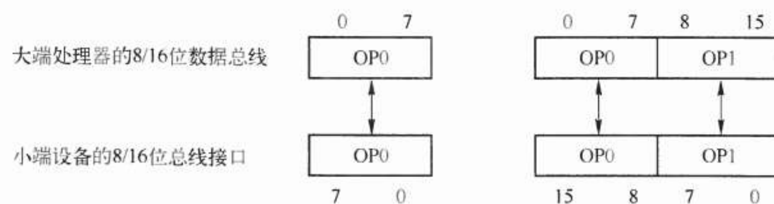


图4-10 大端处理器与小端处设的8/16位连接

PowerPC处理器可以使用sth，stb和lhz，lbz指令对8、16位的外部设备进行访问，并将数据存放在相应的通用寄存器中。当访问结束后，存放在通用寄存器的字节序，即OP0、OP1必须要和存放在小端外设的字节序一致。

PowerPC处理器访问8位的小端外设时，一次只能访问8位数据，如果处理器使用stw或者lwz指令访问8位的小端设备内的32位数据时，数据总线将OP0，OP1，OP2和OP3依次传递到PowerPC的通用寄存器中。

PowerPC处理器对16位的小端外设进行访问时，一次只能访问16位数据，如果处理器使用stw或者lwz指令访问16位的小端设备内的32位数据时，数据总线将OP0～1和OP2～3依次传递到PowerPC的通用寄存器中。

由以上分析，我们可以发现PowerPC处理器使用sth或者lhz指令访问16位的小端设备时，16位的小端设备将数据的第15～0位，传递到PowerPC处理器的总线的第0～15位，然后再将数据传递给相应的通用寄存器。

有许多读者会感到困惑，因为为了保证软件的一致性，PowerPC处理器使用lhz指令访问16位的小端设备的16位寄存器时，需要将结果保存在通用寄存器的第16～31位，而不是0～15位。究竟PowerPC处理器是如何将系统总线中0～15位的数据搬移到寄存器的第16～31位中的呢？为此我们需要对lhz指令进行分析。

lhz指令的描述如下：

```
lhz rD,d(rA)
if rA = 0 then b ← 0
else b ← (rA)
EA ← b + EXTS(d)
rD ← (16)0 || MEM(EA, 1)
```

由此得知 lhz 指令将来自数据总线上的 OP0 与 OP1 直接存入寄存器的第 16～31 位，而将第 0～15 位直接清零。

PowerPC 处理器使用 stb 或者 lbz 指令访问 8 位的小端设备时，8 位的小端设备将数据的第 7～0 位，传递到 PowerPC 处理器的总线的第 0～7 位，然后再将数据最终传递给相应的通用寄存器，lbz 指令的描述如下所示：

```
lbz rD,d(rA)
if rA = 0 then b ← 0
else b ← (rA)
EA ← b + EXTS(d)
rD ← (24)0 || MEM(EA, 1)
```

由此得知 lhz 指令将来自数据总线上的 OP0 直接存入寄存器的第 24～31 位，而将第 0～23 位清零。sth 和 stb 的指令描述与 lhz 和 lbz 类似，本书不再详述。

本节描述了大端处理器的 32 位、16 位及 8 位数据总线与 32 位、16 位和 8 位小端设备进行连接的端模式处理。如果大端处理器的数据总线需要同时支持小端设备的 32 位、16 位及 8 位的数据传送方式，端模式的处理将会更加复杂。IC 的设计人员在设计 PCI 总线桥片的时候会遇到这一类问题，此时设计人员使用多路总线开关解决这一问题。端模式问题的解决需要软硬件协调处理，并在指令集上加以支持。对于小端处理器而言，需要使用软件转换的方法实现大小端模式的匹配；对于大端处理器而言，在外部数据总线与小端外设的连接时必须考虑数据总线连接的拓扑结构。

第 5 章　Linux PowerPC 的进程管理与调度

对于多任务操作系统，进程管理与调度至关重要。在 Linux 系统中，进程管理与调度处于核心地位。Linux 系统支持多用户和多进程，每个用户可以执行多个进程。在一个处理器系统中，CPU 资源最为重要。Linux 系统进程调度与管理的主要功能是协调多个进程使用 CPU 资源。Linux 系统是一个分时操作系统，在 Linux 系统中，每一个 CPU 的运行时间都被划分为一段一段的时间片，各个进程如何合理地使用这些时间片直接影响到整个 Linux 系统的效率。

在 Linux 系统中，一个进程的执行时间可能是几秒钟，也可能是几个月，但是这些进程无论其执行时间长短都要经历若干状态，都有一个完整的生命周期。进程在 Linux 系统中执行过程中的状态转换如图 5-1 所示。

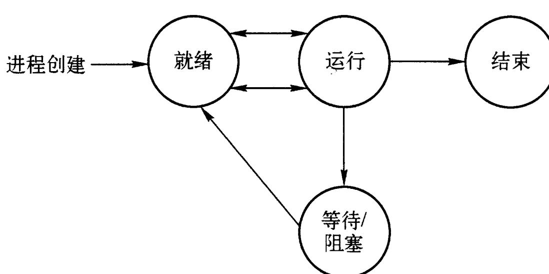


图 5-1　进程状态转换图

Linux 系统使用系统调用 fork、vfork 与 clone 创建进程或者线程。一个进程被创建后将首先进入就绪态，之后等待合适的时机获得 CPU 资源，进入运行态运行。

进程进入运行态后可以执行程序代码。如果进程在执行过程中因为某些资源未能满足不能继续执行，进程将进入等待/阻塞状态。

Linux 调度程序可以将进程从运行态置回就绪态，如进程使用完自己的时间片，或者进程主动放弃 CPU。进程运行完毕后，将进入结束态，结束本次运行。在 Linux 系统中，进程至少要经过就绪、运行和结束状态。在实际情况中，一个进程的状态转换比图 5-1 中的描述要复杂得多。

自从诞生以来，Linux 系统从来没有停止对进程管理与调度算法的改进。Linux 系统的 2.0，2.2，2.4 以及 2.6 内核在进程管理与调度算法上有较大的不同。对于 Linux 2.6 内核的不同版本，如 Linux 2.6.9 和 Linux 2.6.20 内核，其进程管理与调度算法也有较大的不同。一般来说，版本较新的 Linux 内核往往会对之前版本中不太完善的代码进行调整，因此后续版本的设计更加合理一些，为此本章将以 Linux 2.6.20 内核为例介绍进程的管理与调度算法。在 Linux 系统中一个进程包含以下几个必要的组成部分。

(1) 进程描述符。在 Linux 系统中，每个进程都对应一个唯一的进程描述符，进程描述符使用 task_struct 结构。task_struct 结构是 Linux 系统中最重要的结构，该结构包含一个进程使用的所有操作系统信息。

task_struct 结构在进程创建时建立，该结构中包含了多种参数，这些参数记录当前进程使用的各种资源，如与进程调度有关的资源、内存描述符、文件系统描述符以及当前进程打开的文件句柄等一系列资源。

程序员对整个 Linux 系统建立全面的认识之后，才有可能真正了解 task_struct 结构，该结构是 Linux 进程管理与调度模块的核心数据结构。

(2) 进程的正文段。在 Linux 系统中,每一个进程都必须有自己的正文段,即一段可以执行的程序代码。这段程序代码可以被多个进程共享。

(3) 进程的数据段。在 Linux 系统中,每一个进程都必须有自己的数据空间,这段数据空间包括进程在运行过程中使用的全局变量。一个进程的数据段可以被其他进程共享。

(4) 进程的堆栈。每一个进程都需要堆栈存放一些临时数据并利用堆栈实现函数调用。在 Linux 系统中,一个用户进程具有两个堆栈:用户堆栈与核心堆栈,而核心进程只拥有核心堆栈。

在 Linux 系统中,除了进程描述符外,还设置了许多其他结构,如进程运行队列 RQ(Run Queue)和进程等待队列 WQ(Wait Queue)。这些数据结构用来支持进程状态的转换并管理 Linux 系统中的进程。

Linux 系统在进行引导时,需要将这些结构进行必要的初始化,之后依次创建各类进程,并将这些进程放入进程运行队列中,最后等待进程调度程序激活整个 Linux 系统的运行。在 Linux 系统中,这个进程调度程序为 schedule 函数。

Linux 系统在完成中断或者异常事件的处理后,决定是否应该执行 schedule 函数。如果 Linux 系统不支持 Preemptible,用户进程在核心空间进入中断或者异常处理程序时,中断或者异常处理程序返回后,Linux 系统必须回到该进程继续执行;如果一个用户在用户空间进入中断或者异常处理程序时,中断或者异常处理程序返回后,Linux 系统可以根据进程的优先级选择合适的进程获得 CPU 资源。

如果 Linux 系统支持 Preemptible,那么无论进程是在用户空间,或者核心空间进入中断及异常处理程序,中断及异常处理程序返回后,Linux 系统都将根据进程的优先级选择合适的进程获得 CPU 资源。

对于 Linux 系统,系统时钟异常最为重要。该异常会定时产生,并将对当前进程与时间相关的参数进行处理。该异常处理结束后,可能会调用 schedule 函数(如当前进程耗尽 Linux 系统分配的时间片)进行进程调度。

在 Linux 系统初始化后,整个系统处于一个相对静止的状态。此时 Linux 系统初始化完毕其子系统使用的各种资源,然后等待系统时钟异常协调这些资源,并选择合适的进程运行,从而使整个 Linux 系统正常运转。由此可见,整个 Linux 系统由系统时钟异常激活,该异常也被称为整个 Linux 系统的脉搏。

Linux PowerPC 的进程管理与其他 Linux 系统进程管理的绝大多数代码类似。在 Linux PowerPC 的进程管理模块中只有一小部分代码与体系结构相关。本章将详细介绍 Linux PowerPC 的进程管理与调度模块的进程描述符、进程分类、进程的状态转换、系统时钟异常及进程的调度。

5.1 Linux 系统的进程描述符

Linux 系统使用 task_struct 结构描述进程。Linux 系统通过操纵进程描述符对当前进程进行管理与调度。Linux 系统的 task_struct 结构较为复杂,本节仅介绍一些与进程管理与调度联系较为紧密的数据成员,如进程的属性、调度策略、进程间的关系等其他属性,并尽可能对此加以详细描述。尽管如此,绝大多数读者还是不能深入了解 task_struct 结构。读者可能需

要阅读完本书的全部内容后，才能了解 task _ struct 结构中的部分属性，在阅读本章内容时，需要与 Linux 源代码进行对照，以了解这部分内容的细节。

Linux 系统的进程描述符包含了与进程调度、内存管理、文件系统、信号机制、中断系统相关的一些参数，一共有一百多个，本书不可能将其完全解释清楚。尽管如此，本章还是需要从进程描述符开始对 Linux 系统的进程管理与调度进行说明。task _ struct 结构的定义在 ./include/linux/sched.h 文件中。

5.1.1 与进程管理相关的属性

进程描述符包含了一系列参数对进程进行管理，其主要参数有 state、pid、flags、thread 和 thread _ info 等等，其定义如下所示：

```
struct task _ struct {
    volatile long state;
    pid _ t pid;
    pid _ t tgid;
    struct signal _ struct * signal;
    unsigned long flags;
    struct thread _ info * thread _ info;
    struct thread _ struct thread;
    struct linux _ binfmt * binfmt;
    long exit _ state;
    int exit _ code, exit _ signal;

    struct mm _ struct * mm, * active _ mm;
```

1. state 参数

state 参数描述当前进程的运行状态。Linux 系统为进程定义了以下几个状态，这些状态的定义在 ./include/linux/sched.h 文件中。

(1) TASK _ RUNNING。当进程的 state 参数为 TASK _ RUNNING 时，表示该进程处于运行状态或者就绪状态。在 Linux 系统中，进程处于运行或就绪状态时，state 参数都为 TASK _ RUNNING。为区分处于运行状态或者就绪状态的进程，Linux 系统使用了一个全局指针 current。该指针指向正在运行的进程描述符，其定义在 ./include/asm-powerpc/current.h 文件中。

Linux PowerPC 使用通用寄存器 GPR2 保存 current 指针以加快访问速度，current 指针的详细定义如下所示：

```
register struct task _ struct * current asm ("r2");
```

(2) TASK _ INTERRUPTIBLE。当进程的 state 参数为 TASK _ INTERRUPTIBLE 时，表示当前进程处于等待状态。当前进程为了等待某些信号而主动地放弃 CPU 时，可以将 state 参数赋值为 TASK _ INTERRUPTIBLE，并将当前进程从进程运行队列中移出，使进程进入等待状态。当进程等待的信号来临后，进程状态将被改变为 TASK _ RUNNING，同时将该进程放入进程运行队列中。

(3) TASK _ UNINTERRUPTIBLE。当进程的 state 参数为 TASK _ UNINTERRUPTIBLE 时,表示当前进程处于处于等待状态。但是与 TASK _ INTERRUPTIBLE 状态不同,处于该状态的进程不能被信号唤醒,而只能由中断事件唤醒。

在 Linux 系统中,有些设备驱动程序在等待某些中断事件时,可以主动地将进程状态设置为 TASK _ UNINTERRUPTIBLE,使当前进程进入等待状态。当外部中断到来后,该进程状态将被改变为 TASK _ RUNNING,等待调度程序对该进程进行调度。

(4) TASK _ STOPPED。当进程的 state 参数为 TASK _ STOPPED 时,表示 Linux 系统终止了当前进程的运行,即进程终止。进程终止是指 Linux 系统暂时停止进程的运行,然后等待合适的时机重新恢复进程的运行。Linux 系统使用信号机制实现进程的终止。

进程接收到 SIGSTOP、SIGSTP、SIGTTIN 或 SIGTTOU 信号后,当前进程进入终止状态;当进程接收到 SIGCONT 信号后将进程状态改变为 TASK _ RUNNING 状态,等待调度程序对该进程进行调度。Linux 系统使用 ptrace 软件对进程进行调试时,进程状态首先被改写为 TASK _ STOPPED。此外 Linux 系统中,在进程的创建和终止时也使用该状态作为过渡。

(5) TASK _ TRACED。当进程描述符中的 state 参数为 TASK _ TRACED 时,表示当前进程已被 ptrace 系统调用所控制,处于调试状态。

(6) EXIT _ ZOMBIE。当进程的 state 参数为 EXIT _ ZOMBIE 时,表示当前进程已经运行结束但其进程描述符仍然没有被释放。该状态为进程结束前的过渡状态。处于该状态的进程将等待父进程调用 wait4 或者 waitpid 系统调用释放该进程的描述符。

(7) EXIT _ DEAD。当进程的 state 参数为 EXIT _ DEAD 时,表示当前进程最终完成,该状态是进程生命周期的最后一个状态。进程处于该状态时表示父进程正在调用 wait4 或者 waitpid 系统调用结束当前进程。

(8) TASK _ NONINTERACTIVE。当进程的 state 参数为 TASK _ NONINTERACTIVE 时,表示当前进程具有一定的非交互式特征,该参数是最近在 Linux 2.6 版本中加入的。进程处于该状态并不意味着当前进程为非交互式进程。Linux 系统使用宏定义 TASK _ INTERACTIVE 判断一个进程是否为交互式进程,有关交互式进程的详细说明请参见本章下文。

Linux 系统只有 pipe _ wait 函数中使用了该状态,该状态与 TASK _ INTERRUPTIBLE 联合使用。

(9) TASK _ DEAD。当一个进程的 state 参数为 TASK _ DEAD 时,表示当前进程已经执行完毕,但是该进程不需要父进程使用 wait4 或者 waitpid 系统调用回收。

Linux 系统使用宏 set _ task _ state 和 set _ current _ state 改变当前进程描述符的状态。其中宏 set _ task _ state 用来改写进程描述符的状态,而宏 set _ current _ state 用来改写当前正在运行的进程描述符的状态。这两个宏的源代码在 ./include/asm-powerpc/system.h 文件中,如下所示:

```
#define set_task_state(tsk, state_value) set_mb((tsk)->state, (state_value))
#define set_current_state(state_value) set_mb(current->state, (state_value))

#define set_mb(var, value) do { var = value; mb(); } while (0)
```

2. pid,tgid,pgrp 和 signal 参数

在 Linux 系统中,每一个进程都有唯一的标志符。在进程创建时,Linux 系统为每一个进

程分配一个唯一的号码，这个号码叫做进程 ID 号，这个进程 ID 号一般被称为 PID 号。Linux 系统使用进程描述符中的 pid 参数保存 PID 号。

PID 号从 0 开始，直到 Linux 系统允许的最大 PID 进程号。通常 PID 号较小的进程是系统的核心进程和守护进程。这些进程在 Linux 系统引导时即被创建，并且只要 Linux 系统还在运行，它们就处于活跃状态。当用户对进程进行管理时，就必须用到 PID 号。在 Linux 系统中，用户可以使用"ps"命令查看 Linux 系统中的所有进程。

Linux 系统中，除了有进程 PID 号外，还有几种与进程有关的 ID 号，分别是线程组 ID (Thread Group Leader Process)，进程描述符中使用 tgid 参数描述此参数；组 ID(Group Leader Process)，进程描述符使用 signal→pgrp 参数描述此参数；Session ID(Session Leader Process)，进程描述符使用 signal→session 参数描述此参数。

Linux 系统对不同的进程和线程进行分组管理。其中每一个组都有一个组头，tgid 参数就是用来保存这个组头的，而 Session ID 与 Linux 系统的多用户有关。对多进程编程有经验的读者也许使用过进程描述符的 tgid 和 pgrp 参数；而对多用户编程有经验的读者可能会熟悉 Session 参数。本书将不对这些 ID 号详细讨论。这类进程 ID 号主要是提供给 Linux 系统的应用程序，还有一些与进程管理有关的系统调用需要这类进程 ID 号。

在 Linux 系统中，进程 PID 号是一个整型变量，这个整型变量与一个数据结构相关联，这个数据结构也被称作 PID 描述符。Linux 系统使用 pid 结构与 PID 号对应。在 Linux 核心程序中，经常使用 pid 结构，而不是 PID 号对进程进行操作，如下例所示：

进程 P1 使用系统调用 kill 向另外一个进程 P2 传递信号时，必须使用进程 PID 描述符。首先系统调用 kill 使用 P2 的 PID 号作为输入参数，之后 Linux 内核使用该 PID 号作为索引找到该进程 PID 号对应的 PID 描述符，从而找到该进程的进程描述符，然后再作相应的信号传递操作。

pid 结构的详细描述如下所示：

```
struct pid
{
    atomic_t count;
    int nr;
    struct hlist_node pid_chain;
    struct hlist_head tasks[PIDTYPE_MAX];
    struct rcu_head rcu;
};
```

- nr 参数用来存放进程 ID 号。
- pid_chain 参数将 Linux 系统中所有的 PID 描述符组成一个双向链表。Linux 系统使用 hlist_node 结构存放这个双向链表，而不是 list_node 结构。与 list_node 结构相比，hlist_node 结构占用的空间相对较小。
- tasks 参数指向相应的进程描述符，tasks 参数共有 PIDTYPE_MAX 个数据，分别指向与 pid，tgid，pgrp 和 session ID 对应的进程描述符。

为提高查询 PID 描述符搜索速度，Linux 系统将所有 PID 号 HASH 值相同的 PID 描述符连接在一起形成多个双向链表，并使用一个全局数组 pid_hash 保存这些全局链表，其结构如

图 5-2 所示。

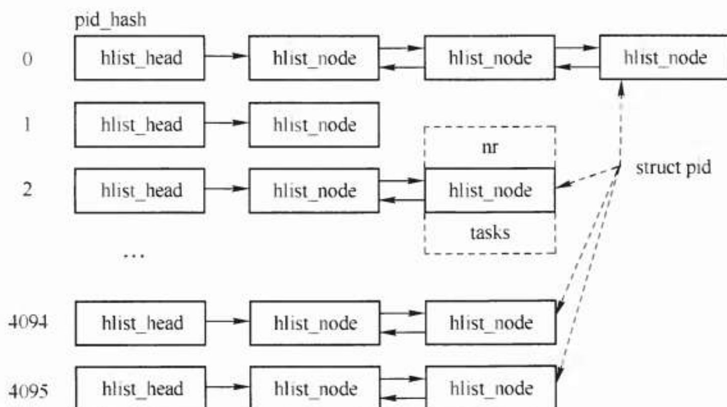


图 5-2 PID描述符结构图

Linux 系统的全局数组 pid _ hash 中一共有 4096 个 Entry，每个 Entry 指向一个由 pid 结构组成的双向链表。全局指针 pid _ hash 在 pidhash _ init 函数中初始化，其源代码如下所示：

```
void _init pidhash_init(void)
{
  int i, pidhash_size;
  unsigned long megabytes = nr_kernel_pages >> (20 - PAGE_SHIFT);

  pidhash_shift = max(4, fls(megabytes * 4));
  pidhash_shift = min(12, pidhash_shift);
  pidhash_size = 1 << pidhash_shift;

  pid_hash = alloc_bootmem(pidhash_size * sizeof( * (pid_hash)));
  if (! pid_hash)
    panic("Could not alloc pidhash! \n");
  for (i = 0; i < pidhash_size; i++)
    INIT_HLIST_HEAD(&pid_hash[i]);
}
```

pidhash _ init 函数由 start _ kernel 函数在 Linux 系统引导时调用，此时 Linux 系统的内存管理尚未初始化完毕，全局数组 pid _ hash 只能使用 Linux 系统中的 Boot Memory 空间，因此在这段程序中使用 alloc _ bootmem 函数分配 pid _ hash 使用的物理地址空间。

pidhash _ init 函数将全局数组 pid _ hash 的每一个 Entry 赋为 NULL，即 hlist _ head→first = NULL。此后，Linux 系统可以使用 alloc _ pid 函数创建 pid 结构，并根据进程的 PID 号的 HASH 值将相应的 pid 结构加入到 pid _ hash 数组中。Linux 系统使用 pid _ hashfn 函数产生 ID 号 HASH 值，该函数在 ./kernel/pid.c 文件中。

```
#define pid_hashfn(nr) hash_long((unsigned long)nr, pidhash_shift) ⇔
#define pid_hashfn (nr) ((nr * (0x9e370001)) >> 20)
```

从密码学的角度上说，pid _ hashfn 函数的 HASH 强度非常低，密码分析员可以轻松地找出其数据碰撞的规律，但是这个算法比较容易实现。当处理器的乘法运算较慢时，程序员也可

以使用以下算法快速实现 pid _ hashfn 函数。

$$pid_hashfn = (x * (2^{31} + 2^{29} - 2^{25} + 2^{22} - 2^{19} - 2^{16} + 2^{0})) >> 20$$

Linux 系统使用 alloc _ pid 函数创建进程的 PID 号和与其相关的 PID 描述符,该函数的源代码如下所示:

```
struct pid *alloc_pid(void)
{
  struct pid *pid;
  enum pid_type type;
  int nr = -1;

  pid = kmem_cache_alloc(pid_cachep, GFP_KERNEL);
  if (!pid)
     goto out;

  nr = alloc_pidmap(current->nsproxy->pid_ns);
  if (nr < 0)
     goto out_free;

  atomic_set(&pid->count, 1);
  pid->nr = nr;
  for (type = 0; type < PIDTYPE_MAX; ++type)
     INIT_HLIST_HEAD(&pid->tasks[type]);

  spin_lock_irq(&pidmap_lock);
  hlist_add_head_rcu(&pid->pid_chain, &pid_hash[pid_hashfn(pid->nr)]);
  spin_unlock_irq(&pidmap_lock);
out:
   return pid;
out_free:
   kmem_cache_free(pid_cachep, pid);
   pid = NULL;
   goto out;
}
```

alloc _ pid 函数的执行流程如下所示:

(1) 调用 kmem _ cache _ alloc 函数从 Linux 系统的 SLAB 分配器中为 pid 结构分配内存空间,有关 kmem _ cache _ alloc 函数和 SLAB 分配器的详细说明见本书的 7.3 节。

(2) 调用 alloc _ pidmap 函数获得进程的 PID 号。

(3) 初始化 pid 结构中的 count 参数和 tasks 参数。并将刚获得的进程 PID 号赋值到 pid -> nr 参数,从而将进程的 PID 号和 pid 结构联系在一起。

(4) 根据进程 PID 号,将 pid 结构加入到 pid _ hash 数组合适的位置中。

在进程的 PID 号和与其相关的 PID 描述符创建完毕后,Linux 系统可以使用 find _ pid 函数根据进程的 PID 号获得其 PID 描述符。

find _ pid 函数首先使用 pid _ hashfn 函数对进程 PID 号进行 HASH。该 HASH 函数的结

构有可能会产生碰撞。当发生碰撞时，find _ pid 函数必须逐个遍历全局数组 pid _ hash 的 pid 结构链表，直到查找到与进程 PID 号完全相同的进程描述符。该函数的源代码如下所示：

```
struct pid * fastcall find_pid(int nr)
{
  struct hlist_node *elem;
  struct pid *pid;

  hlist_for_each_entry_rcu(pid, elem,
       &pid_hash[pid_hashfn(nr)], pid_chain) {
    if (pid->nr == nr)
       return pid;
  }
  return NULL;
}
```

3. flags 参数

除了 state 参数之外，Linux 的进程描述符还使用 flags 参数对进程属性进行额外描述。Linux 在 ./include/linux/sched.h 文件中定义了 flags 参数的各个数据位。

- PF _ ALIGNWARN。flags 参数的 PF _ ALIGNWARN 位为 1 时，表示当前进程正在打印"对齐"警告信息。只有 Intel 80486 体系结构的处理器才使用该参数。
- PF _ STARTING。flags 参数的 PF _ STARTING 位为 1 时，表示当前进程正在被创建。Linux PowerPC 没有使用该参数。
- PF _ EXITING 。flags 参数的 PF _ EXITING 位为 1 时，表示进程正在退出。
- PF _ FORKNOEXEC。flags 参数的 PF _ FORKNOEXEC 位为 1 时，表示进程刚刚被创建，但是还没有执行。
- PF _ SUPERPRIV 。flags 参数的 PF _ SUPERPRIV 位为 1 时，表示当前进程正在使用超级用户权限。
- PF _ DUMPCORE。flags 参数的 PF _ DUMPCORE 位为 1 时，表示当前进程正在 dump 系统信息。
- PF _ SIGNALED。flags 参数的 PF _ SIGNALED 位为 1 时，表示当前进程被 Linux 系统中的某个信号中止。
- PF _ MEMALLOC。flags 参数的 PF _ MEMALLOC 位为 1 时，表示当前进程正在申请物理内存。在 Linux 系统中，内存的管理与分配十分复杂，在绝大多数情况下，Linux 内核使用 kmalloc 函数时，并没有真正地从物理内存池里获得实际物理内存，只有 Linux 内核使用 alloc _ pages 函数时，才会使用物理内存中进行物理内存申请，此时当前进程将 flags 参数符值为 PF _ MEMALLOC。
- PF _ FLUSHER，flags 参数的 PF _ FLUSHER 位为 1 时，表示 Linux 系统正在执行 pdflush 进程。
- PF _ USED _ MATH，flags 参数中的 PF _ USED _ MATH 位为 1 时，表示当前进程正在使用芯片内部的 FPU 处理器，Linux PowerPC 没有使用该参数。
- PF _ FREEZE。flags 参数的 PF _ FREEZE 位为 1 时，表示当前当前进程正在被冻结。

在 Linux 系统中，用户可以使用 freeze _ processes 和 cancel _ freezing 函数冻结和解冻当前进程。Linux 系统被挂起时，使用 freeze _ processes 函数冻结系统中所有的进程；Linux 系统从挂起到重新执行时，使用 cancel _ freezing 函数解冻所有被冻结的进程。

- PF _ NOFREEZE，flags 参数的 PF _ NOFREEZE 位为 1 时，表示该进程不能被冻结。Linux 系统必须等待所有进程的状态不为 PF _ NOFREEZE 后才能够被挂起。
- PF _ FROZEN。flags 参数中的 PF _ FROZEN 位为 1 时，表示当前当前进程已经被冻结。
- PF _ FSTRANS。flags 参数的 PF _ FSTRANS 位为 1 时，表示当前进程正在被 xfs 文件系统使用。xfs 文件系统是 SGI 开发的一个 64 位的高性能文件系统。
- PF _ KSWAPD。flags 参数的 PF _ KSWAPD 位为 1 时，表示当前正在运行的进程为 kswapd 守护进程。
- PF _ SWAPOFF。flags 参数的 PF _ SWAPOFF 位为 1 时，表示当前进程为 swapdoff 进程，swapdoff 进程用来关闭 SWAP 区。
- PF _ LESS _ THROTTLE。flags 参数的 PF _ LESS _ THROTTLE 位为 1 时，表示 nfsd 进程正在处理 nfs 协议请求。
- PF _ BORROWED _ MM，flags 参数的 PF _ BORROWED _ MM 位为 1 时，表示当前进程没有自己的 mm _ struct 结构，而是借用其他进程的 mm _ struct。

在 Linux 系统中，flags 参数中还有其他位，如 PF _ RANDOMIZE，PF _ SWAPWRITE，PF _ SPREAD _ PAGE，PF _ SPREAD _ SLAB，PF _ MEMPOLICY 和 PF _ MUTEX _ TESTER 等。本书不再对这些位一一描述。Linux 系统规定只有当前进程处于运行状态时才可以改变自己的 flags 参数，如以下源代码将当前进程 current 的 flags 参数中的 PF _ EXITING 位置为 1：

```
struct task_struct *tsk = current;
tsk->flags |= PF_EXITING;
```

4. thread _ info 参数

进程描述符的 thread _ info 参数是指向 thread _ info 结构的指针。Linux PowerPC 中，thread _ info 结构的定义在 ./include/asm-powerpc/thread _ info.h 文件中，其详细描述如下：

```
struct thread_info {
    struct task_struct *task;
    struct exec_domain *exec_domain;
    int  cpu;
    int  preempt_count;
    struct restart_block restart_block;
    unsigned long  local_flags;
    unsigned long  flags  __cacheline_aligned_in_smp;
};
```

- task 参数。该参数指向当前进程的 task _ struct 结构。Linux PowerPC 使用该参数纪录当前进程的 task _ struct 结构。
- exec _ domain 参数。该参数用来记录当前进程的正文段的信息。exec _ domain 结构的定义在 ./include/linux/personality.h 文件中。

- cpu 参数。该参数用来记录当前进程在 SMP 处理器的哪个 CPU 中运行，对于单处理器内核系统，该参数为 0。
- restart _ block 参数。该参数用来保存当前进程的 restart _ block 函数。
- local _ flags 参数用来保存 Linux 进程描述符的一些私有数据。Linux PowerPC 中没有使用 local _ flags 参数。
- flags 参数。该参数描述当前进程的一些额外属性，如 TIF _ SYSCALL _ TRACE，TIF _ NOTIFY _ RESUME，TIF _ SIGPENDING，TIF _ NEED _ RESCHED 等其他属性，对此有兴趣的读者可以在 ./include/asm-powerpc/thread _ info.h 文件中找到所有这些属性的定义。在这些属性中 TIF _ NEED _ RESCHED 位最为重要，当 thread _ info 参数中的 flags 参数的 TIF _ NEED _ RESCHED 位为 1 时，Linux 需要对该进程重新进行调度。在 Linux 系统中，程序员可以使用宏 set _ thread _ flag，clear _ thread _ flag，test _ and _ set _ thread _ flag，test _ and _ clear _ thread _ flag 和 test _ thread _ flag 对 flags 参数进行操作；使用宏 set _ need _ resched 和 clear _ need _ resched 对 flags 参数的 TIF _ NEED _ RESCHED 位进行操作。
- preempt _ count 参数。该参数用来记录当前进程是否可以被其他进程抢占。当 Linux 系统支持抢占式内核时，该参数有意义。

所谓抢占式内核是指用户进程在 Linux 内核中运行时可以被其他进程抢占。在 Linux 系统的最初版本中不支持抢占式，当用户进程在 Linux 内核中运行时不可以被其他进程抢占。进程在这种 Linux 系统中运行时，虽然中断或者异常处理程序可以切换在 Linux 内核中运行的进程，但是当中断或者异常处理程序结束后，还需要返回到该进程重新执行，而不能切换到其他进程中执行。抢占式内核允许 Linux 系统切换在内核中运行的进程。为此，Linux 系统设置了 preempt _ count 参数。该参数可以被宏 preempt _ disable，preempt _ enable 设置。宏 preempt _ disable 可以对 preempt _ count 参数进行递增，而宏 preempt _ enable 可以对 preempt _ count 参数进行递减。

宏 preempt _ disable，preempt _ enable 的源代码在 ./include/linux/preempt.h 文件中，其源代码如下：

```
#define preempt_disable() \
do { \
  inc_preempt_count(); \
  barrier(); \
} while (0)

#define preempt_enable() \
do { \
  preempt_enable_no_resched(); \
  barrier(); \
  preempt_check_resched(); \
} while (0)
```

为实现抢占式内核，Linux 系统规定当 preempt _ count 参数等于 0 时，该进程可以被抢占；该参数大于 0 时不可抢占；该参数小于 0 时，表示系统出现了 Bug。

本书不对抢占式内核做详细描述。抢占式内核的概念实际上比较容易理解,有许多操作系统在设计之初就支持抢占式内核。Linux 系统的 2.4 版本的后期才支持抢占式内核,因此一些读者对于抢占式内核有些陌生。

在 Linux 系统中,抢占式内核的实现较为复杂,这是因为在 Linux 系统的设计之初并不支持抢占式内核,有许多核心代码在书写时并没有考虑到在 Linux 内核中执行的进程可以被抢占,因此在 Linux 系统中引入抢占式内核的概念时需要重新编写这些代码。

一个进程的 thread _ info 结构在进程的 task _ struct 创建之后建立。Linux 系统使用宏 alloc _ thread _ info 和 free _ thread _ info 申请和释放 thread _ info 结构。在 Linux PowerPC 中,其代码如下所示:

```
#define alloc_thread_info(tsk) \
        ((struct thread_info *)__get_free_pages(GFP_KERNEL, THREAD_ORDER))

#define free_thread_info(ti) free_pages((unsigned long)ti, THREAD_ORDER)
```

在 Linux PowerPC 中,THREAD _ ORDER 的值为 1;GFP _ KERNEL 是_ get _ free _ pages 函数申请物理空间使用的参数。

_ get _ free _ pages 函数在物理内存中分配一个或者多个物理地址连续的页面。执行 alloc _ thread _ info 函数时,THREAD _ ORDER 参数为 1,此时 Linux 系统将为数据结构 thread _ info 分配两个物理地址连续的页面空间。对于 32 位的 E500 内核,Linux PowerPC 的一个物理页面的大小为 4 KB,因此 Linux 系统将为数据结构 thread _ info 分配一个物理地址连续的 8 KB 空间。由 thread _ info 结构可以发现,thread _ info 不会需要 8 KB 这么大的物理空间。Linux 系统为 thread _ info 设置 8 KB 大小的空间的目的是让进程的核心堆栈与进程描述符的 thread _ info 共享这段 8 KB 大小的空间。

在 Linux 系统中,用户进程共有两个堆栈,用户堆栈和核心堆栈。用户进程在 Linux 系统的用户空间中执行时,使用用户堆栈;用户进程通过系统调用进入 Linux 内核空间运行时,使用核心堆栈。

在 Linux 系统中,每一个进程都有自己的核心堆栈,这个核心堆栈将与该进程描述符的 thread _ info 参数共享同一段内存空间。Linux 系统的进程描述符、thread _ info 结构和进程的核心堆栈的关系如图 5-3 所示。

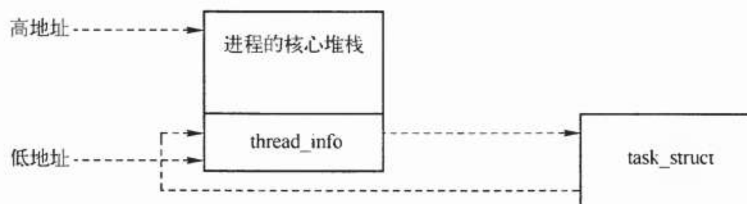


图 5-3　进程描述符、thread _ info 与进程核心堆栈结构图

进程的核心堆栈与 thread _ info 共享一个 8 KB 物理地址连续的区域。其中进程的核心堆栈的增长方向从高地址到低地址,当进程的核心堆栈与 thread _ info 所占用的空间重叠时,将引起进程的核心堆栈溢出。

进程进入到内核执行时,可以通过当前进程描述符的 thread _ info 参数获得该进程核心堆

栈的栈顶指针。

5. thread 参数

thread 参数是指向 thread _ struct 结构的指针。thread 参数保存了一些与处理器体系结构相关的寄存器。这些寄存器包括通用寄存器,进程的堆栈指针等其他寄存器信息。进行进程切换时,Linux 系统需要将 PowerPC E500 内核中的寄存器与 thread _ struct 结构中保存的值进行互换。

Linux PowerPC 使用 SPRG3 寄存器保存 thread 参数以加快对此结构的访问速度。在基于 E500 内核的 PowerPC 处理器中,SPRG3 寄存器保存 thread 参数的虚拟地址;在其他体系的 PowerPC 处理器中,如基于 603E 内核,SPRG3 寄存器保存 thread 参数的物理地址。因为在这类 PowerPC 处理器中,进入中断或者异常处理程序时,MMU 将会被关闭,因此必须通过在 SPRG3 寄存器中保存的物理地址才能获得有关进程的一些信息。

在 Linux PowerPC 中,thread _ struct 结构的定义在 ./include/asm-powerpc/processor.h 文件中。对于 PowerPC E500 内核,thread _ struct 结构的主要数据成员如下所示:

```
struct thread_struct {
    unsigned long   ksp;            /* Kernel stack pointer */
    struct pt_regs  *regs;          /* Pointer to saved register state */
    mm_segment_t fs;                /* for get_fs() validation */
    void    *pgdir;                 /* root of page-table tree */
    signed long last_syscall;
    unsigned long   dbcr0;          /* debug control register values */
    unsigned long   dbcr1;
    unsigned long   dabr;
};
```

- ksp 参数指向进程核心堆栈的栈顶指针。
- regs 参数保存当前进程使用的寄存器。
- pgdir 参数存放当前进程 page table 的基地址。在 Linux 系统中,每一个用户进程都有独立的 3GB 虚拟地址空间,这些进程地址空间彼此独立。
- last _ syscall 参数用来保存系统调用号。

Linux 系统创建子进程时,thread 参数将被初始化。

6. mm 和 active _ mm 参数

mm 参数保存当前进程的内存地址空间。在 Linux 系统中,用户进程具有自己的内存地址空间,而核心进程将使用 Linux 核心的内存地址空间,没有自己的内存地址空间,因此核心进程的 mm 参数为空。

active _ mm 参数用来保存进程的用户内存地址空间。用户进程的 active _ mm 参数指向用户的内存地址空间。而核心进程没有用户地址空间。在 Linux 中,核心进程将借用用户进程的内存地址空间。

一个用户进程进入 Linux 内核运行时,如果该进程使用 kernel _ thread 函数创建了一个新的核心进程,则此核心进程的 active _ mm 参数指向用户进程的内存地址空间。Linux 系统的第一个核心进程 init _ task 在创建时,其 active _ mm 参数指向 Linux 内核中的全局变量init _ mm。

由上所述，用户进程的 mm 和 active _ mm 参数相等；核心进程的 mm 参数为 NULL，active _ mm 参数将借用前一个用户进程的 active _ mm 参数。核心进程被切换时借用的用户进程地址空间将被释放，即该进程的 active _ mm 参数被置为 NULL。本书在进程切换一节中还要继续讨论 mm 和 active _ mm 参数。读者可能需要完成本书的全部内容后，才可能理解 mm 和 active _ mm 这两个参数。

Linux 系统采用 mm 和 active _ mm 这两个参数保存进程的内存地址空间有一些深层次的原因。对于 32 位处理器，普通进程运行在用户空间时，可以访问的用户地址空间大小为 3 GB；普通进程运行在 Linux 核心空间时，可以同时访问核心的 1 GB 空间和用户的 3 GB 用户地址空间；核心进程运行只能运行在 Linux 核心空间，此时可以访问核心的 1 GB 空间和借用的 3 GB 用户地址空间。

因此在 Linux 内核中运行的进程都可以有一个完整的 4 GB 空间。这个完整的 4 GB 空间，对处理 Linux 系统中的异常事件至关重要。Linux 系统使用 mm _ struct 结构描述 mm 和 active _ mm 参数，mm _ struct 结构是 Linux 内存管理系统中最重要的数据结构，本书将在第 7 章详细介绍 mm _ struct 结构。

7. binfmt 参数

Linux 系统中支持许多种可执行文件格式，其中每一种可执行文件都与一种数据结构相对应。Linux 系统中支持的主要可执行文件格式主要有 aout，elf，elf _ fdpic 和 flat 格式。在 binfmt 参数中存放着加载这些可执行文件的函数如 load _ binary 和 load _ shlib 函数。

8. exit _ state，exit _ code 和 exit _ signal 参数

exit _ state 与 exit _ code 参数用来存放进程结束的状态和返回值，这些值将由子进程传递给父进程，并由父进程处理这些参数。exit _ signal 参数存放子进程结束时产生的中止信号，父进程将处理这个结束信号，并释放子进程使用的进程描述符和其他与子进程相关的全部资源。

5.1.2 与进程调度有关的属性

Linux 属于分时操作系统，将 CPU 的运行时间分成若干个时间片，每一个进程使用完毕自己的时间片后放弃 CPU，由其他进程继续使用 CPU。在多个进程使用一个 CPU 资源时，Linux 系统需要进行进程调度。

Linux 系统采用基于优先权的调度方式，可以根据进程的特点对其采用不同的调度策略，还可以根据进程的优先权确定进程的运行时间片，以保证 Linux 系统各个进程合理地使用 CPU 资源。其中，公平与高效是 Linux 系统进程管理与调度模块所追求的主要目标。

Linux 系统将进程分为以下几类：

（1）交互式进程。交互式进程是需要处理大量交互式信息的进程，比如处理键盘、鼠标等输入信息的进程。这些进程的共同特点是需要较长时间等待用户输入信息，而一旦这些信息通过中断或者其他方式获得后，需要迅速获得 CPU 资源。这类进程包括文本编译器、shell 处理程序等等。

（2）批处理进程。这类进程基本上不需要用户交互信息，可以放入后台运行。这类进程的优先权较低，但是需要持续不断地使用 CPU 资源。科学计算、文件编译、数据库查找等进程属于这一类进程。

（3）实时进程。这类进程用来在规定的时间内对实时信息进行处理。这类进程的优先权

高于以上两类进程。

(4) 普通进程。不属于上述进程的分类的进程。在 Linux 系统中,大多数进程都是普通进程。普通进程和交互式进程可以动态转换。

普通进程在 CPU 中运行时可以根据与用户的交互频繁程度调整为交互式进程,反之亦然。普通进程与批处理进程或者实时进程进行转换需要使用系统调用。

Linux 的进程管理与调度根据以上分类,在进程描述符中设置了许多参数管理这几类进程。这些参数是进程管理与调度的核心,包括与进程优先权相关的参数如 static _ prio、normal _ prio 和 prio,与进程使用的时间片相关的参数如 time _ slice、first _ time _ slice 等。在进程描述符中,与进程调度相关的属性如下所示:

```
int prio, static_prio, normal_prio;
struct list_head run_list;
struct prio_array *array;
unsigned long rt_priority;
unsigned short ioprio;
unsigned long sleep_avg;
unsigned long long timestamp;
unsigned long long sched_time;   /* sched_clock time spent running */

enum sleep_type sleep_type;
unsigned long policy;
unsigned int time_slice, first_time_slice;
spinlock_t pi_lock;
struct plist_head pi_waiters;
```

1. static _ prio 参数

static _ prio 参数表示当前进程的静态优先权。在 Linux 系统中,程序员只能通过系统调用 nice 修改一个进程的静态优先权。如果在进程运行的过程中,没有使用系统调用对此参数进行修改,此参数将一直保持不变,static _ prio 参数在进程的运行期间不会因为其占用 CPU 的时间长短而发生变化。

子进程的 static _ prio 参数继承其父进程的 static _ prio 参数。如果在进程的创建过程中,没有任何父进程改写过 static _ prio 参数的值,那么 static _ prio 参数的值将为 Linux 系统的第一个核心进程,即 init _ task 进程的 static _ prio 参数值,这个值为 MAX _ PRIO - 20,即为 120。用户可以通过系统调用 nice 修改进程的 static _ prio 参数。在 Linux 系统中,系统调用 nice 将调用 set _ user _ nice 函数修改当前进程的 static _ prio,set _ user _ nice 函数的源代码在 ./kernel/sched.c 文件中。其主要源代码如下:

```
void set_user_nice(struct task_struct *p, long nice)
{
  struct prio_array *array;
  int old_prio, delta;
  unsigned long flags;
...
```

```
        p->static_prio = NICE_TO_PRIO(nice);
        set_load_weight(p);
        old_prio = p->prio;
        p->prio = effective_prio(p);
    ...
    }
```

set_user_nice 函数具有两个参数 p 和 nice。其中参数 p 用来指向当前进程的进程描述符,参数 nice 是一个 long 类型的数值,其范围为-20~19,用来设置进程的 static_prio。

set_user_nice 函数使用以下步骤修改 static_prio 参数。

(1) 调用 NICE_TO_PRIO 将进程的 static_prio 参数改写为 120 + nice,因此一个进程的 static_prio 参数在 100~139 之间。

(2) 调用 effective_prio 函数将进程的 prio,normal_prio 参数与 static_prio 的值进行同步。prio 和 normal_prio 参数将在下文介绍。

set_user_nice 函数的实现比上述过程要复杂得多。但是在现阶段,读者先暂时这样理解该函数。在 Linux 系统中, static_prio 参数是一个进程的基本属性。许多与调度相关的属性是根据 static_prio 参数计算得出的。

2. rt_priority

Linux 系统支持实时进程,并使用 rt_priority 参数描述当前进程的实时优先权。在进程运行过程中,程序员可以通过系统调用修改进程的 rt_priority 参数。

在 Linux 系统中,进程在创建初期都是普通进程,此时进程的 rt_priority 参数没有意义。但是这些普通进程可以通过系统调用 sched_setscheler 和 sched_setparam 改写进程的 rt_priority 属性和调度策略将普通进程改为实时进程。sched_setscheler 系统调用可以改写进程的进程的 rt_priority 参数和调度策略,而 sched_setparam 系统调用只能改写该进程的 rt_priority 参数。

sched_setscheler 和 sched_setparam 系统调用最终需要调用_setscheduler 函数来更改进程的 rt_priority 参数和进程调度属性,_setscheduler 函数的源代码在 ./kernel/sched.h 文件中,其源代码如下:

```
static void _setscheduler(struct task_struct *p, int policy, int prio)
{
  BUG_ON(p->array);

  p->policy = policy;
  p->rt_priority = prio;
  p->normal_prio = normal_prio(p);
  /* we are holding p->pi_lock already */
  p->prio = rt_mutex_getprio(p);
  /*
   * SCHED_BATCH tasks are treated as perpetual CPU hogs:
   */
  if (policy == SCHED_BATCH)
    p->sleep_avg = 0;
  set_load_weight(p);
}
```

_ setscheduler 函数具有三个参数 p，policy 和 prio。参数 p 用来指向当前进程的进程描述符，参数 policy 用来设置进程的调度策略，而参数 prio 用来设置进程的 rt _ priority。

进程的 rt _ priority 参数在 0～99 之间。当 rt _ priority 越大时，进程的实时优先权级越高。rt _ priority 参数与进程的调度策略参数相关，只有采用了 SCHED _ FIFO 和 SCHED _ RR 调度策略的进程，其 rt _ priority 参数才有意义。

在 Linux 系统中，成为实时进程需要具备两个条件：一是当前进程的调度策略，即 policy 参数的值，为 SCHED _ FIFO 或者 SCHED _ RR；二是当前进程的 rt _ priority 值有效。

一个进程的 rt _ priority 参数与 prio，normal _ prio 参数密切相关。Linux 系统修改此参数之后，需要使用 normal _ prio 和 rt _ mutex _ getprio 函数同步进程的 prio 和 normal _ prio 参数。

3．prio 参数

在进程描述符中，prio 参数的作用至关重要。Linux 系统使用基于优先权的调度策略，其中这个优先权与 prio 参数中直接相关。在 Linux 系统中，prio 参数一个动态变量。当出现以下四种情况时，prio 参数将会改变：

（1）当进程的 static _ prio 参数发生变化时。

（2）当程序员使用_ setscheduler 函数将 Linux 系统中的一个普通进程更改为实时进程时，prio 参数将随 rt _ priority 参数的变化而变化。

（3）一个普通进程在执行过程中，会根据其在 CPU 中的睡眠时间，动态地调整 prio 参数。一个进程在 CPU 中的睡眠的时间越长，其 prio 参数的值越小。

（4）进程优先权的继承。进程优先权继承将在下文详细描述。

Linux 的调度程序根据 prio 参数确定程序的优先权，一个进程的 prio 参数越大表示该进程的优先权越低，prio 参数的取值范围为 0～139。当一个进程的 prio 的值为 0～99 时，此进程为实时进程，当 prio 的值为 100～139 时，此进程为普通进程。prio 参数的值需要根据当前进程是实时进程还是普通进程分别讨论。

对于实时进程，程序员使用_ setscheduler 函数改变进程的 rt _ priority 参数和调度策略的同时也会将 prio 参数更新为 rt _ mutex _ getprio(p)。在大多数情况下，这个值与进程的 normal _ prio 参数相同为 MAX _ RT _ PRIO-1 - p->rt _ priority，即 99- p->rt _ priority。在 Linux 系统中，宏 MAX _ RT _ PRIO 等于 100。

实时进程的 prio 参数在运行过程中不随运行时间的长短而发生变化，实时进程的 prio 参数将始终保持不变，而不随进程在 CPU 的平均睡眠时间而改变。如果程序员需要对实时进程的优先权 prio 参数进行调整时，必须使用 sched _ setscheler 或者 sched _ setparam 系统调用才能改变 prio 参数。

当一个进程为普通进程时，prio 参数的计算较为复杂。在进程创建时，prio 参数将继承父进程的 normal _ prio 参数的值。这里提醒读者注意：在 Linux 2.6 的后期版本创建新进程时使用 normal _ prio 参数而不是 prio 参数将父进程的优先权传递给子进程。在绝大多数情况下，normal _ prio 参数与 prio 参数的值相同。本书将在下文介绍 normal _ prio 参数。

普通进程的 prio 参数随着进程在 CPU 中的平均睡眠时间的长度而发生变化。所谓平均睡眠时间是指进程在 CPU 中睡眠的平均值，Linux 系统的进程描述符使用 sleep _ avg 参数记录此值。普通进程根据 sleep _ avg 参数，为普通进程设立一个临时变量 bonus，bonus 的值与进程的 sleep _ avg 参数有关。

普通进程使用_ normal _ prio 函数计算 prio 参数和 bonus 变量。_ normal _ prio 函数的定义在 ./kernel/sched.c 文件中。

```
static inline int _normal_prio(struct task_struct *p)
{
  int bonus, prio;

  bonus = CURRENT_BONUS(p) - MAX_BONUS / 2;

  prio = p->static_prio - bonus;
  if (prio < MAX_RT_PRIO)
      prio = MAX_RT_PRIO;
  if (prio > MAX_PRIO-1)
      prio = MAX_PRIO-1;
  return prio;
}
```

在这段程序中,首先调用宏 CURRENT _ BONUS(p) - MAX _ BONUS / 2 获得当前进程的 bonus 参数。其中 MAX _ BONUS 的值为 bonus 变量可能的最大值,该值为 10。因此 bonus 变量的值为 CURRENT _ BONUS(p)-5。之后这段程序根据进程的 static _ prio 参数调整当前进程的 prio 参数,并保证所得到的 prio 参数在 100~139 之间。宏 CURRENT _ BONUS 和一些与其相关的宏的定义在 ./kernel/sched.c 文件中。

```
#define CURRENT_BONUS(p) \
    (NS_TO_JIFFIES((p)->sleep_avg) * MAX_BONUS / MAX_SLEEP_AVG)

#define MAX_BONUS (MAX_USER_PRIO * PRIO_BONUS_RATIO / 100)

#define MAX_SLEEP_AVG (DEF_TIMESLICE * MAX_BONUS)

#define MAX_USER_PRIO (USER_PRIO(MAX_PRIO))

#define USER_PRIO(p) ((p)-MAX_RT_PRIO)

#define DEF_TIMESLICE (100 * HZ / 1000)
#define MAX_BONUS (MAX_USER_PRIO * PRIO_BONUS_RATIO / 100)

#define PRIO_BONUS_RATIO 25
```

根据这些宏定义,我们可以通过简单的运算,获得宏 MAX _ BONUS 和宏 CURRENT _ BONUS(p)的计算公式。宏 MAX _ SLEEP _ AVG 与 MAX _ BONUS 有关,我们在这里也给出宏 MAX _ SLEEP _ AVG 的计算公式。

```
MAX_BONUS = (MAX_USER_PRIO * PRIO_BONUS_RATIO / 100)
      = ((USER_PRIO(MAX_PRIO)) * 25) / 100
      = (MAX_PRIO - MAX_RT_PRIO) * 25 /100
      = 40 * 25 / 100 = 10

CURRENT_BONUS(p) = NS_TO_JIFFIES((p)->sleep_avg) * 10 / HZ
      = ((p->sleep_avg) / (1,000,000,000 / HZ)) * 10 / HZ
      = p->sleep_avg/100,000,000 (ns) = p->sleep_avg/ 100 (ms)
```

```
MAX_SLEEP_AVG = DEF_TIMESLICE * MAX_BONUS
        = (100 * HZ / 1000) * MAX_BONUS
        = (100 * HZ / 1000) * 10
        = 1HZ
```

这些计算公式使用的“HZ”表示 Linux 系统时钟的频率，即每秒钟 Linux 系统可以产生多少次系统时钟异常。“HZ”可以在“make menuconfig”命令配置 Linux 源代码时初始化。在 Linux PowerPC 中，HZ 可以被设置为 100，250 或者 1000。

通过以上计算，可以发现 bonus 变量和 prio 参数的值与 sleep_avg 参数相关。在_normal_pri 函数中的 bonus 变量与 prio 参数的值如下所示：

```
bonus = p->sleep_avg/100 - 5;
prio = p->static_prio - bonus;
```

由以上这些公式，可以得出以下结论：

- 进程的 sleep_avg 参数在 0～99 ms 内时，bonus 为 −5，prio 参数为 static_prio + 5。
- 进程的 sleep_avg 参数在 100～199 ms 内时，bonus 为 −4，prio 参数为 static_prio + 4。
- 进程的 sleep_avg 参数在 200～299 ms 内时，bonus 为 −3，prio 参数为 static_prio + 3。

  ……

- 进程的 sleep_avg 参数在 800～899 ms 内时，bonus 为 3，prio 参数为 static_prio −3。
- 进程的 sleep_avg 参数在 900～999 ms 内时，bonus 为 4，prio 参数为 static_prio −4。
- 进程的 sleep_avg 参数等于 1000 ms 时，bonus 为 5，prio 参数为 static_prio −5。注意在 Linux 系统中进程的 sleep_avg 参数不会大于 1000 ms。

由此我们可以发现一个普通进程的 prio 参数可以在 static_prio−5～static+5 之间变化，但是无论 prio 参数如果变换，其值一定在 100～139 之间。同时我们也可以发现，对于两个普通进程，只要进程 P1 的 static_prio 参数大于进程 P2 的 static_prio 参数 + 10，那么在 Linux 系统中，进程 P1 的优先权一定大于进程 P2 的优先权。

4．normal_prio，pi_lock 和 pi_waiters 参数

normal_prio，pi_lock 和 pi_waiters 参数的主要作用是处理进程的优先权更迭（Priority Inversion）。Linux 系统使用优先权继承（Priority Inherit）方法处理进程的优先权更迭。在一个实时系统中，发生进程优先权更迭的可能性更大。在 Linux 系统中，优先权继承这一方法主要针对实时进程。

normal_prio 参数用来保存进程的正常优先权。pi_waiters 参数的意义将在下文详细介绍，而 pi_lock 参数用来保护对 pi_waiters 参数的访问。在介绍 pi_waiter 参数之前，我们首先需要对进程的优先权更迭和优先权继承这两个概念进行简单说明。

假设在一个支持进程抢占的操作系统中运行着三个进程，分别为进程 A，B 和 C，其中进程 A 的优先权最高，进程 B 的优先权次之，而进程 C 的优先权最低。在一般情况下，进程 A 能够对进程 B 进行抢占，而进程 B 能够对进程 C 进行抢占，而不会发生进程 B 抢占进程 A 的情况。但是在某种情况下，进程 B 能够通过抢占进程 C 的执行而间接抢占进程 A。如图 5-4 所示。

图 5-4 中一共有 3 个进程，分别为 A，B，C。其中进程 A 的优先权最高，进程 B 次之，而进程 C 的优先权最低。假定进程 A，B 和 C 运行在一个允许抢占的操作系统中，并按照以下顺序执行：

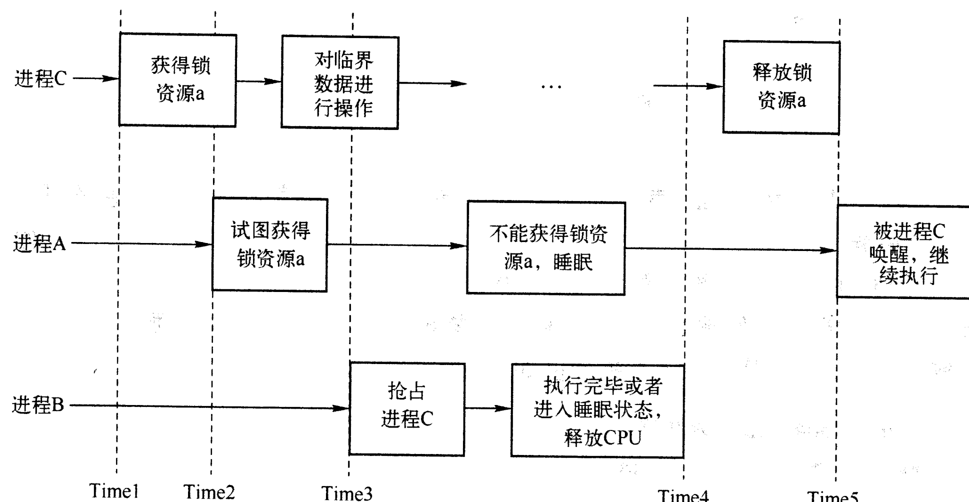


图 5-4　优先权更迭的例子

(1) 进程 C 首先获得 CPU 资源，在 Time1 时刻开始运行，并获取一个互斥锁 a，之后访问互斥锁保护的临界数据。

(2) 在 Time2 时刻，进程 A 开始执行，此时由于进程 A 的优先权大于进程 C 的优先权，因此进程 A 将抢占进程 C 使用的 CPU 资源，并执行进程 A 的程序。如果进程 A 也需要访问互斥锁 a 保护的临界段，由于互斥锁 a 已经被进程 C 使用，因此进程 A 将进入睡眠状态，等待进程 C 释放互斥锁 a。

(3) 而在 Time3 时刻，进程 B 开始执行，此时进程 C 并没有释放互斥锁 a，因此进程 A 不会被唤醒。此时在当前系统中进程 B 的优先权最高，于是进程 B 将抢占进程 C 使用的 CPU 资源，并执行进程 B 的程序。

(4) 进程 A 将在系统中等待，直到进程 B 执行完毕或者进入睡眠状态释放 CPU 资源后，才能继续执行。

(5) 在 Time4 时刻，进程 B 释放 CPU 资源，此时进程 C 将获得 CPU 资源，继续执行并释放互斥锁 a。此时进程 A 被唤醒，继续执行程序。

从以上过程我们可以发现，由于进程 A 在等待进程 C 释放互斥锁 a 而一直处于睡眠状态。此时其他进程，只要其优先权大于进程 C 的优先权(尽管其优先权可能小于进程 A 的优先权)，就可以抢占进程 C 的运行，事实上也抢占了进程 A 的运行。这种低优先权的进程可以抢占高优先权的进程运行的行为，被称为优先权更迭。

在操作系统设计过程中，系统程序员可以采取许多种方法解决这种优先权更迭的问题。采用优先权继承的方法可以有效地解决这种“优先权更迭”的问题。

所谓“优先权继承”，是指低优先权的进程可以暂时获得较高的优先权，以避免被其他进程抢占，从而有效地避免了“优先权更迭”的发生。下文将以 Linux 系统为例简要说明“优先权继承”的实现。

由上文可知，优先权较高的进程在没有获得互斥锁进行休眠时，有可能发生“优先权更迭”的现象。为此 Linux 系统设置了专门的互斥锁 rt_mutex 处理“优先权更迭”。程序员可以使用 rt_mutex_lock 获得互斥锁 rt_mutex，使用 rt_mutex_unlock 函数释放互斥锁 rt_mutex。

使用 rt _ mutex _ lock 函数获得互斥锁 rt _ mutex 时，需要做一系列的检查。如果调用 rt _ mutex _ lock 函数的进程，其优先权大于 pi _ waits 链表中进程的最大优先权，则使用当前进程的优先权，覆盖在 pi _ waits 参数中等待进程的最大优先权。Linux 系统中，可能存在多个进程等待同一个互斥锁 rt _ mutex 被释放，每当一个新的进程需要获得此互斥锁时，将可能提高当前拥有此互斥锁 rt _ mutex 进程的优先权 prio，以避免"优先权更迭"。

使用 rt _ mutex _ unlock 函数释放互斥锁 rt _ mutex 时，将使用 rt _ mutex _ lock 函数提高的进程优先权 prio 还原。

下文将结合本节发生"优先权更迭现象"的实例，说明 Linux 系统如何采用"优先权继承"的方法避免"优先权更迭现象"的发生。

(1) 进程 C 首先获得 CPU 资源，在 Time1 时刻开始运行，并获取一个 rt _ mutex 类型的互斥锁 a，之后访问对互斥锁保护的临界数据。

(2) 在 Time2 时刻，进程 A 开始执行，此时由于进程 A 的优先权大于进程 C 的优先权，因此进程 A 将抢占进程 C 使用的 CPU 资源，并执行进程 A 的程序。此时进程 A 调用 rt _ mutex _ lock 函数试图获得互斥锁 a，并将进程 C 的优先权提升，使其等效于进程 A 的优先权。由于互斥锁 a 正在被进程 C 使用，于是进程 A 将进入睡眠状态，等待进程 C 释放互斥锁 a。

(3) 而在 Time3 时刻，进程 B 开始执行。此时进程 B 的优先权小于进程 A、C 的优先权，因此进程 B 将不会获得 CPU 资源。此时进程 B 将在系统中等待。

(4) 在 Time4 时刻，进程 C 释放互斥锁 a，此时进程 C 的优先权将被恢复。之后进程 A 而不是进程 B 将获得 CPU 资源，继续执行。

(5) 在 Time5 时刻，进程 A 释放 CPU 资源，进程 B 将获得 CPU 资源，开始执行。该过程见图 5-5。

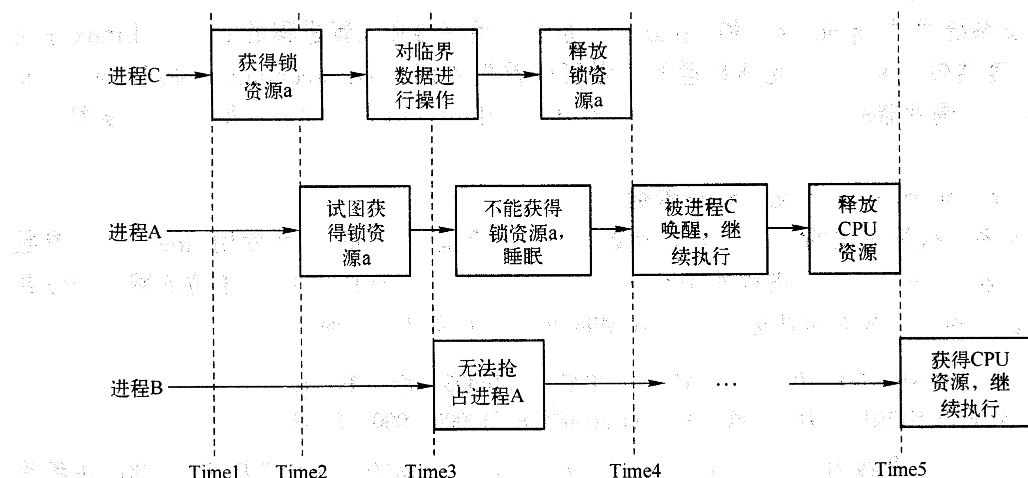


图 5-5　优先权继承的例子

由以上分析，我们发现采用"优先权继承"的方法可以有效解决"优先权更迭"问题。但是使用"优先权继承"的方法时，进程 A 的优先权，即 prio 参数，将会被暂时提高。为此 Linux 系统设置了 normal _ prio 参数，当 prio 参数因为使用"优先权继承"方法暂时改变了 prio 参数时，normal _ prio 参数依然保持不变。

Linux 系统创建子进程时，其子进程的 prio 参数从 normal _ prio 参数中获得，而不使用父进程的 prio 参数，从而有效地避免了将父进程中已经被提升的 prio 参数传递给子进程。

本节中有关"优先权继承"的内容，仅仅讲述了一些最基本的一些知识。Linux 系统的 rt _ mutex 实现机制比较复杂，本书不再详细这部分源代码。

Linux 系统有关 rt _ mutex 的源代码在 ./kernel/remutex.c 和 ./include/linux/rtmutex.h 文件中。对此有兴趣的读者可以结合 ./Documentation/rt-mutex-design.txt 文件，详细了解有关 Linux 系统 rt _ mutex 的实现。

5. ioprio 参数

ioprio 参数存放进程的 I/O 优先权。在 Linux 2.6.13 版本之后的内核支持进程的 I/O 优先权，系统程序员可以通过修改 /sys/block/<device>/queue/scheduler 文件设置<device>设备的 I/O 调度器。如以下代码将 hda，即硬盘设备 hda 的 I/O 调度器，设置为 CFQ。

```
# echo cfq > /sys/block/hda/queue/scheduler
```

目前 Linux 系统只支持 CFQ(Completely Fair Queuing)类型的 I/O 调度器。CFQ 定义了以下三种级别的 I/O 类型：

- IOPRIO _ CLASS _ RT。ioprio 参数为此值时，该进程访问 I/O 设备时具有最高的优先级。
- IOPRIO _ CLASS _ BE。ioprio 参数为此值时，表示系统采取一种相对公平的策略访问相应的 I/O 设备，并保证所有的进程都有机会访问到 I/O 设备。这种类型也是 ioprio 参数使用的缺省值。
- IOPRIO _ CLASS _ IDLE。ioprio 参数为此值时，表示该进程只有在其他进程都不访问此 I/O 设备时，才能对此 I/O 设备进行访问。

Linux 系统通过 ioprio _ get 和 ioprio _ set 系统调用获得和设置进程的 ioprio。Linux 系统中 I/O 调度器较为复杂，对此感兴趣的读者可以首先阅读 ./documentation/block/ioprio.txt 文件获取 I/O 调度器的一些入门知识，之后再分析 ./fs/ioprio.c 文件以获得 I/O 调度器的实现细节知识。

6. time _ slice，first _ time _ slice 参数

Linux 系统根据进程的 static _ prio 参数为进程分配运行时间片，而使用 time _ slice 参数保存这个"运行时间片"。当进程使用 CPU 时，"运行时间片"将以 Jiffy 为单位递减。Jiffy 是 Linux 系统中的一个基本的时间单位，Jiffy 和时间(ns)的换算如下所示：

```
#define NS_TO_JIFFIES(TIME) ((TIME) / (1000000000 / HZ))
#define JIFFIES_TO_NS(TIME) ((TIME) * (1000000000 / HZ))
```

当 time _ slice 参数为 0 时，表示 Linux 系统分配给该进程的时间片消耗完毕，此时进程将会很快地让出 CPU 资源。在进程使用完当前的时间片后，Linux 系统将该进程放入就绪队列，然后为该进程重新分配一段"运行时间片"。Linux 系统将会选择调度时机重新将该进程运行，然后周而复始，直到该进程运行结束。

init _ task 进程是 Linux 系统中第一个进程，该进程的 time _ slice 参数在 Linux 系统初始化时被静态初始化。读者在 ./include/linux/init _ task.h 文件的宏 INIT _ TASK 中找到有关 time _ slice 参数的定义。

```
#define INIT_TASK(tsk)                                    \
{
    .state = 0,                                           \
    .thread_info = &init_thread_info,                     \
    .usage = ATOMIC_INIT(2),                              \
    .flags = 0,                                           \
    .lock_depth = -1,                                     \
    .prio = MAX_PRIO-20,                                  \
    .static_prio = MAX_PRIO-20,                           \
    .normal_prio = MAX_PRIO-20,                           \
    .policy = SCHED_NORMAL,                               \
    .cpus_allowed = CPU_MASK_ALL,                         \
    .mm = NULL,                                           \
    .active_mm = &init_mm,                                \
...
    .time_slice = HZ,                                     \
}
```

Linux系统使用宏 INIT_TASK 初始化 init_task 的进程描述符。由上文所示，time_slice 参数为 HZ 个 Jiffies。Linux 系统中使用 HZ 为 init_task 进程的 time_slice 赋值有时会引起读者的误解，因为 HZ 的本意用来表示 Linux 系统时钟的频率。因此直接使用 HZ 从逻辑上说类型并不匹配。Linux 系统赋予 init_task 进程的执行时间是 1 秒钟，因此使用宏 NS_TO_JIFFIES(1,000,000,000)定义 init_task 进程的 time_slice 参数会更准确，也更容易理解一些。

在 Linux 系统中，除了 init_task 进程，其他进程都是由其父进程创建出来的，这些进程都要使用 sched_fork 函数初始化 time_slice 参数。在 sched_fork 函数中，父子进程将平分父进程的 time_slice 参数。以下代码中 p 表示子进程，current 表示父进程。sched_fork 函数中与 time_slice 参数有关的代码如下所示：

```
p->time_slice = (current->time_slice + 1) >> 1;
p->first_time_slice = 1;
current->time_slice >>= 1;
if (unlikely(!current->time_slice)) {
    current->time_slice = 1;
    scheduler_tick();
}
```

由上述代码，读者可以发现 sched_fork 函数将父进程的 time_slice 参数的一半分给子进程的 time_slice 参数。

当父进程的 time_slice 参数为 1 时，通过上述代码计算出子进程的 time_slice 参数为 1，而父进程将使用完毕自己本次的时间，其 time_slice 参数为 0。此时 sched_fork 函数将调用 scheduler_tick 函数将父进程需要重新调度标志 TIF_NEED_RESCHED 赋值为 1，然后选择合适时机使父进程让出 CPU。有关 scheduler_tick 函数的详细说明，请参考 5.4.2 节。

在创建子进程的过程中,父进程将自己的时间片分配给子进程,从而保证了父子进程占用的总时间片不变。

first _ time _ slice 参数表示子进程在创建时从父进程获得的时间片是否使用完毕,为 1 表示该进程获得的时间片仍有剩余,为 0 时表示首次获得的时间片已经使用完毕。在子进程初始化时,first _ time _ slice 的值被父进程的 sche _ fork 函数置为 1,并对子进程的 time _ slice 参数赋初值。

当进程首次获得的时间片 time _ slice 随着进程的运行递减为 0 时,Linux 系统将 first _ time _ slice 参数置为 0,随后 Linux 系统的调度程序将很快中止该进程的运行。

Linux 系统设置 first _ time _ slice 参数的主要作用是帮助父进程回收子进程的剩余时间片。有时子进程的运行时间十分短暂,甚至在没有使用完父进程给予的第一个运行时间片,就已经执行完毕。此时在子进程结束并调用 sched _ exit 函数时,需要利用 first _ time _ slice 参数将子进程未用完的时间片回收。

sched _ exit 函数中使用 first _ time _ slice 的详细源代码如下:

```
if (p->first_time_slice && task_cpu(p) == task_cpu(p->parent)) {
    p->parent->time_slice += p->time_slice;
    if (unlikely(p->parent->time_slice > task_timeslice(p)))
            p->parent->time_slice = task_timeslice(p);
}
```

由上述源代码所示,当进程描述符的 first _ time _ slice 为 1,且父子进程在同一个 CPU 中执行时,父进程将回收子进程未使用完的时间片。父进程在回收完子进程剩余的时间片后,如果其剩余时间片大于 task _ timeslice(p)时,则将父进程的剩余时间片的值置为 task _ timeslice(p)。

在 Linux 系统中,进程使用完父进程给予的时间片后,使用 task _ timeslice 函数根据该进程的 static _ prio 参数重新对 time _ slices 参数赋值,task _ timeslice 函数在 ./kernel/sched.c 文件中。其源代码如下所示:

```
static inline unsigned int task_timeslice(struct task_struct *p)
{
     return static_prio_timeslice(p->static_prio);
}

static unsigned int static_prio_timeslice(int static_prio)
{
  if (static_prio < NICE_TO_PRIO(0))   //NICE_TO_PRIO(0) = 120
    return SCALE_PRIO(DEF_TIMESLICE * 4, static_prio);
  else
    return SCALE_PRIO(DEF_TIMESLICE, static_prio);
}

#define SCALE_PRIO(x, prio) \
  max(x * (MAX_PRIO - prio) / (MAX_USER_PRIO / 2), MIN_TIMESLICE)
```

由以上代码,读者可以发现,task _ timeslice 函数的返回值,只与进程的 static _ prio 参数

有关。

- 当 static _ prio 小于 120 时，p->time _ slice = max(20 * (140 - p->static _ prio), 5)
- 当 static _ prio 大于等于 120 时，p->time _ slice = max(5 * (140 - p->static _ prio), 5)

以上关于 time _ slice 参数的说明主要针对 Linux 系统中的普通进程。对于实时进程 time _ slice 参数的值与进程所采用的调度算法有关。当该进程采用 SCHED _ FIFO 算法时，time _ slice 参数的值没有意义，当进程采用 SCHED _ RR 算法时，time _ slice 参数的计算与普通进程相同。

由上分析可以得出：在 Linux 系统中，进程的 static _ prio 参数在很大程度上决定了该进程在系统中运行时间的比例。进程运行时，time _ slice 参数将不断递减，等到该值为 0 时，time _ slice 参数将会被重值。

7. timestamp 参数

进程描述符的 timestamp 参数保存当前进程最近一次使用 CPU 的时间戳。Linux 系统中使用 sched _ clock 函数获得当前时间戳，timestamp 参数是一个 unsigned long long 类型，Linux PowerPC 可以使用 PowerPC 内部的 TB(Time Base Register)或者外部的 RTC 获得当前系统的时间戳。

Linux 系统使用 timestamp 参数计算该进程的睡眠时间以及在 CPU 中的运行时间。在 Linux 系统中，系统程序员可以使用当前 CPU 的时间戳减去该进程的 timestamp 参数获得当前进程的睡眠时间以及当前进程在 CPU 中的运行时间。

8. run _ list 参数和 array 参数

一个进程在加入到进程运行队列 RQ(Run Queue)后，才可以接受 Linux 系统的调度。进程运行队列是进程调度中的一个重要概念，见本书的 5.4.1 节。

进程描述符使用 list _ head 结构描述 run _ list 参数。进程使用 run _ list 参数将其加入到进程运行队列，等待 Linux 系统的进程调度，当进程随眠时，该进程将从进程运行队列中脱离，即从进程运行队列中将进程描述符的 run _ list 参数摘除。

当进程加入到进程运行队列后，array 参数将加入到进程运行队列 RQ 中。当进程为普通进程时，array 参数将指向进程运行队列 RQ→active 队列，当进程为批处理进程时，array 参数将指向进程运行队列 RQ→expired 队列。

这段代码在_ activate _ task 函数中，该函数在 ./kernel/sched.c 文件中。

```
static void _activate_task(struct task_struct *p, struct rq *rq)
{
  struct prio_array *target = rq->active;

  if (batch_task(p))
    target = rq->expired;
  enqueue_task(p, target);
  inc_nr_running(p, rq);
}
```

_ activate _ task 函数首先判断当前进程是否为批处理进程，如果是，则将 target 变量赋值为 rq→active，否则将 target 变量赋值为 rq→expired。之后调用 enqueue _ task 函数将进程 p 加

入到 rq→active 或者 rq→expired 中。enqueue_task 函数在 ./kernel/sched.c 文件中,其源代码详解见 5.4.12 节。

要充分理解进程描述符的 run_list 参数和 array 参数,首先需要深入理解 Linux 进程运行队列的详细结构。在此,读者只要记住进程描述符中包含这两个参数即可。

9. sleep_avg 与 sleep_type 参数

Linux 系统中有一些进程需要较高的交互式级别,如我们在 Linux 系统中经常使用的 shell 进程。shell 进程在绝大部分时间要等待用户的输入命令,在用户输入命令后,shell 进程将启动相应的进程执行,之后继续等待用户再次输入命令。shell 进程需要具有较高的响应速度,但是不需要有太多的运行时间。为了处理这类进程交互式进程,进程描述符中设置了 sleep_avg 和 sleep_type 参数。

sleep_avg 参数表示进程的平均睡眠时间,该值与进程的 prio 参数直接相关。Linux 系统的普通进程,其 sleep_avg 参数越大,prio 参数的值越低。prio 参数的值可以根据 sleep_avg 参数进行动态调整。此外 sleep_avg 参数还与进程的交互式级别有关。

Linux 系统使用 sleep_type 参数保存进程的交互式级别。进程描述符的 sleep_type 参数可以为 SLEEP_NONINTERACTIVE、SLEEP_INTERRUPTED、SLEEP_INTERACTIVE 和 SLEEP_NORMAL,这些参数将在下文详细解释。

在 Linux 系统中,sleep_avg 参数的取值范围为 0～NS_MAX_SLEEP_AVG,其单位为 ns。NS_MAX_SLEEP_AVG 的值的计算如下所示:

```
NS_MAX_SLEEP_AVG
    = (JIFFIES_TO_NS(MAX_SLEEP_AVG))
    = MAX_SLEEP_AVG * 1000000000 / HZ
    = HZ * 1000000000 / HZ
    = 1000000000 (ns)
    = 1 (s)
```

当进程获得 CPU 运行后,sleep_avg 参数的值将不断减少,直到其值为 0。注意,当进程放弃 CPU,进入睡眠状态时,sleep_avg 参数将一直保持不变,而当进程重新获得 CPU,sleep_avg 参数才会被重新计算。重新计算的结果当然是随着进程在 CPU 中睡眠的时间越长,其 sleep_avg 参数的值越大。

再次强调,当进程睡眠时,sleep_avg 参数不会发生变化。这是因为 Linux 系统只能通过 scheduler_tick 函数调整当前进程使用的 sleep_avg 参数,而不能调整处于睡眠状态的进程的 sleep_avg 参数,scheduler_tick 函数是 Linux 进程调度的核心,其详细说明见 5.4.2 节。

sleep_avg 参数的计算较为复杂,下文将对此详细分析。

对于 Linux 系统的 init_task 进程,进程描述符的 sleep_avg 参数没有什么意义,但是对于其他进程的意义重大。父进程创建子进程时,Linux 系统将会调用 do_fork 函数,do_fork 函数将会调用 wake_up_new_task 函数设置子进程的 sleep_avg。wake_up_new_task 函数除了可以设置子进程的 sleep_avg 参数外,还可以完成其他其他操作,该函数的详细说明见 5.3.1 节。wake_up_new_task 函数使用以下公式计算父子进程的 sleep_avg 参数,在以下公式中,p 表示子进程,current 表示父进程。

```
#define CHILD_PENALTY  95
#define PARENT_PENALTY  100

JIFFIES_TO_NS(CURRENT_BONUS(p))
    = JIFFIES_TO_NS (NS_TO_JIFFIES((p)->sleep_avg)
        * MAX_BONUS / MAX_SLEEP_AVG)
    = (p->sleep_avg) * MAX_BONUS / MAX_SLEEP_AVG

p->sleep_avg = JIFFIES_TO_NS(CURRENT_BONUS(p) *
            CHILD_PENALTY / 100 * MAX_SLEEP_AVG / MAX_BONUS);
        = ((p->sleep_avg) * MAX_BONUS / MAX_SLEEP_AVG)) *
          (CHILD_PENALTY / 100 * MAX_SLEEP_AVG / MAX_BONUS))
        = (p->sleep_avg) * CHILD_PENALTY / 100
        = (p->sleep_avg) * 95 /100

current->sleep_avg = JIFFIES_TO_NS(CURRENT_BONUS(current) *
            PARENT_PENALTY / 100 * MAX_SLEEP_AVG / MAX_BONUS)
        = (p->sleep_avg) * PARENT_PENALTY /100
        = p->sleep_avg
```

通过以上公式,我们可以发现父进程创建子进程后,子进程的 sleep_avg 参数将会被改变,而父进程的 sleep_avg 参数保持不变。

在 Linux 系统中,除了 wake_up_new_task 函数,进程调度函数 schedule 和 activate_task 函数(该函数的主要功能是将进程加入到进程运行队列中)也会根据进程在 CPU 中的睡眠时间调整 sleep_avg 参数。这些函数最终都会调用 recalc_task_prio 函数计算进程的 sleep_avg 参数。

recalc_task_prio 函数的源代码在 ./kernel/sched.c 文件中。该函数的主要功能是计算进程描述符的 sleep_avg 参数,然后使用 sleep_avg 参数的值计算进程的 prio 和 normal_prio 参数。该函数的源代码详解如下:

```
static int recalc_task_prio(struct task_struct *p, unsigned long long now)
{
    /* Caller must always ensure 'now >= p->timestamp' */
    unsigned long sleep_time = now - p->timestamp;
```

recalc_task_prio 函数有两个输入参数 p 和 now。p 参数指向进程描述符,now 参数用来记载当前的系统时间。recalc_task_prio 函数的返回值为当前进程的 prio 参数。

recalc_task_prio 函数首先获得进程的睡眠时间 sleep_time,sleep_time 为当前系统的时间戳 now 和进程最近一次使用 CPU 的时间戳,即进程的 timestamp 参数之差。Linux 系统使用这种方法确定一个进程从睡眠到重新被激活之间的时间,即进程的睡眠时间 sleep_time。

在调用 recalc_task_prio 函数时,程序员需要保证 now 参数大于 p→timestamp 参数。

```
if (batch_task(p))
  sleep_time = 0;
```

当一个进程使用 SCHED_BATCH 调度策略时,将 sleep_time 的值置为 0,即认为该进程

没有进行睡眠。此时 recalc _ task _ prio 函数将不会调整进程的 sleep _ avg 参数,因此也不会调整进程的 prio 参数。由此可见采用 SCHED _ BATCH 调度策略的进程不会根据 sleep _ avg 参数调整进程的优先权。因此采用这种调度策略的进程在 CPU 上睡眠得再久,也不会提高其优先权。

```
if (likely(sleep_time > 0)) {
    unsigned long ceiling = INTERACTIVE_SLEEP(p);
```

当进程不使用 SCHED _ BATCH 调度策略时,sleep _ time 的值一定大于 0,此时该函数将计算休眠阈值 ceiling,该阈值的计算公式如下所示:

```
DELTA (p) = SCALE (TASK_NICE (p) + 20, 40, MAX_BONUS) -
              20 * MAX_BONUS / 40 + INTERACTIVE_DELTA
          = SCALE (PRIO_TO_NICE (p->static_prio) + 20, 40, 10) - 5 + 2
          = SCALE (p->static_prio - 120 + 20, 40, 10) -3
          = ((p->static_prio - 100) * 10 /40) -3
          = p->static_prio/4 -25 -3 = p->static_prio/4 -28

ceiling   = INTERACTIVE_SLEEP(p)
          = JIFFIES_TO_NS (MAX_SLEEP_AVG
                * (MAX_BONUS / 2 + DELTA(p) + 1) / MAX_BONUS - 1)
          = JIFFIES_TO_NS (HZ * (10/2 + DELTA(p) + 1) / 10 -1)
          = JIFFIES_TO_NS (HZ * (6 + p->static_prio/4 -28)/10) -1)
          = JIFFIES_TO_NS (HZ * (p->static_prio/4 -22)/10) -1)
```

通过以上公式,可以发现休眠阈值 ceiling 与进程的 static _ prio 参数有关。一个进程 static _ prio 参数的值越小,其对应的休眠阈值 ceiling 越小。当进程的 static _ prio 参数为 139 时,休眠阈值为 1199ms;当进程的 static _ prio 为 120 时,休眠阈值为 799ms;当 static _ prio 为 100 时,休眠阈值为 299。因此 static _ prio 的值越小,sleep _ time 越容易大于相应的休眠阈值 ceiling。

```
if (p->mm && sleep_time > ceiling && p->sleep_avg < ceiling) {
    p->sleep_avg = ceiling;
    p->sleep_type = SLEEP_NONINTERACTIVE;
```

这段程序对 p->mm,sleep _ time > ceiling 和 p->sleep _ avg < ceiling 三个条件进行判断。如果这三个条件同时成立,则 sleep _ avg 参数被调整为休眠阈值 ceiling,并将进程的交互式级别 sleep _ type 置为 SLEEP _ NONINTERACTIVE。这三个条件的含义如下所示:

(1) 进程描述符的 mm 是否为空,当 mm 为空时,表示当前进程是核心进程。

(2) 进程的睡眠时间 sleep _ time 大于休眠阈值 ceiling。

(3) 进程描述符的平均睡眠时间 sleep _ avg 是否小于休眠阈值 ceiling。

当三个条件同时成立时表示当前进程是用户进程,其 sleep _ avg 参数小于休眠阈值 ceiling,且该进程在 CPU 中的睡眠时间 sleep _ time 大于休眠阈值 ceiling。也就是说这是当前用户进程的 sleep _ time 第一次超过休眠阈值 ceiling。

此时 Linux 系统不会将当前进程设置为交互式进程,而是将进程描述符的 sleep _ avg 参

数调整为休眠阈值 ceiling,并将进程的交互式级别改为 SLEEP _ NONINTERACTIVE。

```
} else {
  if (p->sleep_type == SLEEP_NONINTERACTIVE && p->mm) {
    if (p->sleep_avg >= ceiling)
      sleep_time = 0;
    else if (p->sleep_avg + sleep_time >= ceiling) {
        p->sleep_avg = ceiling;
        sleep_time = 0;
    }
  }
  p->sleep_avg += sleep_time;
}
```

当上文中的三个条件有一个不成立时,将执行这段代码。这段程序将首先判断进程的交互式级别 sleep _ type 是否为 SLEEP _ NONINTERACTIVE 以及进程是否为用户进程,如果条件成立,则调整进程的 sleep _ avg 参数,sleep _ avg += sleep _ time。

```
  if (p->sleep_avg > NS_MAX_SLEEP_AVG)
    p->sleep_avg = NS_MAX_SLEEP_AVG;
}
return effective_prio(p);
```

这段程序调整 p->sleep _ avg 参数,保证该参数的取值范围在 0~NS _ MAX _ SLEEP _ AVG 之间。然后调用 effective _ prio 函数调整进程的 normal _ prio 参数。

effective _ prio 函数的返回值也作为 recalc _ task _ prio 函数的返回值。effective _ prio 函数的源代码详解如下:

```
static int effective_prio(struct task_struct *p)
{
  p->normal_prio = normal_prio(p);
  /*
   * If we are RT tasks or we were boosted to RT priority,
   * keep the priority unchanged. Otherwise, update priority
   * to the normal priority:
   */
  if (! rt_prio(p->prio))
    return p->normal_prio;
  return p->prio;
}
```

该函数的执行如下:

(1) effective _ prio 函数调用 normal _ prio 函数计算进程的 normal _ prio 参数,如果当前进程是实时进程,则 normal _ prio = MAX _ RT _ PRIO-1 - p->rt _ priority;否则使用_ normal _ prio 函数计算进程描述符的 normal _ prio 参数,_ normal _ prio 函数的详细说明见上文。

（2）如果当前进程是实时进程，effective_prio 函数将进程描述符的 prio 参数返回，即不改变当前进程的 prio，否则返回已经根据进程的 bonus 参数调整过的 normal_prio 参数。

Linux 系统使用进程描述符的 sleep_avg 和 sleep_type 参数动态地改变进程优先权，这使得一个进程睡眠较长时间后，会获得比较高的动态优先权，以保证 Linux 系统进程调度的公平性。Linux 系统还使用 sleep_avg 参数判断一个进程是否为交互式进程。在 Linux 系统中，交互式进程可以获得更高的响应度。有关交互式进程的内容将在 5.4 节中详细描述。

10. policy 参数

policy 参数存放进程的调度策略。在 Linux 系统中，每个进程可以单独设置自己的调度策略。Linux 系统共支持四种进程调度策略，分别为 SCHED_NORMAL、SCHED_FIFO、SCHED_BATCH 和 SCHED_RR。

（1）SCHED_NORMAL 是 Linux 系统最常见的进程调度策略，Linux 的普通进程都使用这种调度策略。这种调度策略基于进程使用的时间片和进程的优先权进行调度。这种调度策略可以保证 Linux 系统中的每个进程可以轮换使用 CPU 资源。使用这种调度策略时，进程的优先权和进程使用的 CPU 时间片可以根据进程的占用 CPU 的时间而动态改变。

当一个进程长期使用 CPU 资源后，其优先权将会降低。反之一个长期没有得到 CPU 资源的进程，其优先权将会提高。从而 Linux 系统中的进程都有机会获得 CPU 资源运行，使得 Linux 进程调度达到整体最优的效果。

一般来说，对于一个非实时的应用，Linux 系统的所有进程都可以使用这种调度策略。这样 Linux 系统将根据各个进程的优先权来合理地使用 CPU 的时间片，不会有某个进程因为长时间没有得到 CPU 资源而被饿死的现象发生。在 Linux 系统中，普通进程只能使用 SCHED_NORMAL 或者 SCHED_BATCH 策略进行进程调度。

（2）SCHED_BATCH 是 Linux 系统处理批处理进程的调度策略。批处理进程不需要与用户交互，因此，它们经常在后台运行。因为批进程不需要很快被地 CPU 响应，在 Linux 系统中，这类进程常受到调度程序的慢待。典型的批处理进程是程序设计语言的编译程序，数据库搜索引擎及科学计算程序。

（3）SCHED_FIFO 和 SCHED_RR 是实时进程采用的进程调度策略。Linux 系统引入了实时进程的概念，允许将一个进程定义为实时进程，并且规定实时进程的优先权一定大于普通进程。

在 Linux 系统中，实时进程可以采用 SCHED_FIFO 或者 SCHED_RR 策略进行进程调度。其中 SCHED_RR 策略使用时间片轮转的方法调度实时进程。使用 SCHED_RR 调度策略的进程一旦用完进程的时间片就被移动到优先级队列的队尾，并允许同一优先级的其他进程运行。如果在当前系统中没有“同一优先级”的其他进程，该进程将继续运行。

SCHED_FIFO 策略使用先来先服务的方法调度实时进程。使用 SCHED_FIFO 调度策略的实时进程将一直占用 CPU 资源运行。直到该进程被阻塞后或者结束后，其他进程才能获得 CPU 资源运行。

系统程序员在设计实时进程必须要注意到在 Linux 系统中实时进程的优先权一定大于普通进程的优先权。在设计中，系统程序员需要统筹考虑 Linux 系统的整体效率，合理设计实时进程。在 Linux 系统中，如果一个实时进程长时间占用 CPU 资源，其他普通进程都将被饿死。

在 Linux 系统中，普通进程或者交互式进程可以使用 SCHED _ NORMAL 调度策略；实时进程可以使用 SCHED _ FIFO 和 SCHED _ RR 调度策略；而批处理进程可以采用 SCHED _ BATCH 调度策略，批处理进程也是普通进程的一种。

5.1.3 进程描述符的其他属性

在 Linux 中，除了 init _ task 进程外，每个进程都有自己的父进程，有的进程还有子进程和兄弟进程。进程描述符使用 real _ parent，children 和 sibling 数据成员纪录这些关系。Linux 所有的进程组成了一个基于胖树的网状结构。

进程描述符还记录了一些有关系统安全性的属性。Linux 是个多用户系统，这些不同用户还可以组成各自的组。task _ struct 使用 uid，gid，euid，eguid，suid 等其他数据成员表示这些属性。

进程描述符记录了进程使用的与文件系统，文件描述符，内存系统相关的各种信息。进程描述符的结构较为复杂。本书不再对进程描述符的这些属性一一介绍。

操作系统进程调度与管理模块与内存管理和中断系统紧密相连。要求读者对进程调度和管理、内存管理、中断系统、信号机制甚至 Linux 系统中的应用程序都要有较深刻的认识。

进程描述符是进程调度与管理的核心。掌握了进程描述符的细节知识也就掌握了进程调度的核心，在任何一个操作系统中，进程调度和切换模块需要尽可能地缩短自身占用 CPU 的时间。也是因为这个原因，进程调度和切换所使用的算法短小精悍，理解这些算法的难度并不大。

5.2 Linux 系统中的核心进程与普通进程

Linux 系统有许多种进程，如核心进程(Kernel Thread)、用户进程、用户线程等。有些进程在 Linux 内核中创建并运行；有些在用户空间中创建，但是可以在 Linux 内核和应用空间中运行。

这些不同种类的进程在 Linux 中的作用不同。Linux 的系统用户与一般用户可以根据需要创建不同种类的进程。

一个典型的进程一般要包含以下几个部分。

(1) 进程描述符与 PID(Process ID)。在 Linux 系统中使用 task _ struct 结构管理进程描述符。一个进程的进程描述符存放在 Linux 核心内存中，用户进程不能操作进程描述符。在 Linux 系统中，可以使用 PID 号表示进程。Linux 内核可以通过进程的 PID 号查找到相应的进程描述符。

(2) 进程的正文段。用户进程创建时，进程的正文段将被调入到用户空间中；而核心进程使用 Linux 内核的正文段。一个进程的正文段可以被其他进程共享。子进程在创建时可以共享其父进程的正文段空间；也可以在子进程创建完毕后使用系统调用 execve 改变进程的正文段。

(3) 用户进程的数据段和栈段。用户进程的数据段和栈段在进程的用户空间中。

(4) 核心堆栈。在 Linux 系统中，每一个进程在内核空间都有一个独立而且唯一的核心

堆栈。这个堆栈用来支持进程在 Linux 核心中运行。如 5.1.1 节所示，进程的核心堆栈与进程描述符的 thread _ info 参数共享一个 8KB 大小的空间。

5.2.1 核心进程

核心进程是指仅在 Linux 核心空间中运行的进程。核心进程具有进程描述符、进程的正文段和核心堆栈。核心进程的正文段在 Linux 核心空间中，与其他核心进程及 Linux 内核共享相同的核心正文段，核心进程正文段的组成见 ./arch/powerpc/kernel/vmlinux.lds.S 文件。

核心进程的堆栈与进程描述符的 thread _ info 参数共享一个 8 KB 大小的空间，每一个核心进程具有独立的核心堆栈。核心进程不在用户空间含有数据段与栈段。Linux 系统中，核心进程包括使用 kernel _ thread 函数在 Linux 核心中创建的进程和 init _ task 进程。在 Linux 系统中，核心进程可以调用 execve 函数将核心进程更改为用户进程。

1. Linux 系统的 init _ task 进程

init _ task 进程是 Linux 内核中的第一个进程，这个进程贯穿于整个 Linux 系统的初始化，init _ task 进程也是 Linux 系统中唯一没有使用 kernel _ thread 函数创建的核心进程。在 init _ task 进程的执行后期使用 kernel _ thread 函数创建第一个核心进程，并完成 Linux 系统初始化。在 init _ task 进程完成 Linux 系统的初始化后，将退化为 idle 进程，当 Linux 系统中没有其他进程使用 CPU 资源时，该进程将被执行。

Linux 系统在进行初始化时，使用宏 INIT _ TASK 初始化 init _ task 进程，并静态地为 init _ task 进程建立进程描述符、核心栈段、init _ task 进程与 Linux 内核共享正文段。Linux PowerPC 的 init _ task 进程的定义在 ./arch/powerpc/kernel/init _ task.c 文件中，而宏 INIT _ TASK 的源代码在 ./arch/powerpc/kernel/init _ task.h 文件中，其代码如下所示：

```
struct task_struct init_task = INIT_TASK(init_task);

#define INIT_TASK(tsk)                          \
{
  .state   = 0,                                 \
  .thread_info = &init_thread_info,             \
  .flags   = 0,                                 \
  .prio    = MAX_PRIO-20,                       \
  .static_prio = MAX_PRIO-20,                   \
  .normal_prio = MAX_PRIO-20,                   \
  .policy   = SCHED_NORMAL,                     \
  .mm     = NULL,                               \
  .active_mm = &init_mm,                        \
  .time_slice = HZ,                             \
...
  .thread = INIT_THREAD,                        \
...
}
```

由上述代码可见，Linux 系统将 init _ task 进程的进程描述符进行静态赋值。其主要参数

的初始值如下所示:

1) state 参数为 0,即为 TASK _ RUNNING 状态,表示当前进程正在运行或者在就绪队列中等待执行。

2) thread _ info 参数为 init _ thread _ info, init _ thread _ info 的定义如下所示:

```
#define init_thread_info(init_thread_union.thread_info)

union thread_union {
  struct thread_info thread_info;
  unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

init _ thread _ union 是一个 thread _ union 类型的联合结构,在该结构中 thread _ info 参数与 8 KB 大小的 stack 参数共享一个空间。init _ task 进程采用这种方法将其 thread _ info 参数和该进程所需要的核心堆栈进行初始化。

3) prio、static _ prio 和 normal _ prio 参数为 120, policy 参数为 SCHED _ NORMAL。

4) Linux 系统的核心进程没有属于自己的内存地址空间 mm,所有的核心进程的 mm 参数都为 NULL,而 active _ mm 参数借用用户的进程地址空间。内核初始化进程 init _ task 在 Linux 系统进行初始化时的 active _ mm 参数为 init _ mm, init _ mm 变量使用宏 INIT _ MM 进行初始化。宏 INIT _ MM 在 ./include/linux/init _ task.h 文件中定义,该宏将内存描述符的 pgd 参数设置为全局变量 swapper _ pg _ dir, swapper _ pg _ dir 变量是 Linux 内核初始化时使用的 pgd 指针。有关 pgd 指针的详细说明见本书的第 7 章。

5) thread 参数使用宏 INIT _ THREAD 进行初始化,宏 INIT _ THREAD 对 init _ task 进程的 thread 参数进行设置,包括 init _ task 进程使用的堆栈指针, pgd 和其他参数进行设置,其定义在 ./include/asm-powerpc/processor.h 文件中,如下所示:

```
#define INIT_THREAD {                              \
    .ksp = INIT_SP,                                \
    .regs = (struct pt_regs *)INIT_SP - 1,         \
    .fs = KERNEL_DS,                               \
    .fpr = {0},                                    \
    .fpscr = { .val = 0, },                        \
    .fpexc_mode = 0,                               \
}
#define INIT_SP    (sizeof(init_stack) + (unsigned long) &init_stack)
#define init_stack    (init_thread_union.stack)
```

从以上源代码中,可以发现 init _ task 进程使用宏 INIT _ SP 设置核心堆栈的栈顶指针。该值为 init _ thread _ union.stack + 8KB,也等于 init _ thread _ union.thread _ info + 8KB,因为 init _ thread _ union 的 stack 和 thread _ info 参数共享同一段物理地址连续的空间。

在 Linux 系统的初始化阶段, init _ task 进程即被建立,这个进程将对 Linux 系统的进程调度子系统、内存管理子系统、文件管理子系统、信号机制等其他所有的 Linux 子系统进行初始化,然后使用 kernel _ thread 函数创建 init 进程,最后执行 cpu _ idle 函数使处理器处于空闲状

态。在 Linux 系统中,如果没有其他活动进程,Linux 系统将使用 init _ task 进程作为缺省进程运行。

在标准的 Linux 系统中,核心进程 init 是由 init _ task 进程创建的唯一进程,将完成 init _ task 进程未完成的其他 Linux 系统的初始化工作,包括 work _ queue、驱动程序的初始化,释放 Boot Memory 空间等。在核心进程 init 的最后,将执行 execve 系统调用将 init 进程从核心进程转换为用户进程,进一步对 Linux 系统的用户进程进行初始化。Linux 的用户进程都是由核心进程 init 创建或者派生出来的。

2. 其他核心进程

与 Linux PowerPC 中的普通进程相比,核心进程与 Linux 核心共享虚地址空间,因此在核心进程中地址的虚实地址转换效率较高(PowerPC 处理器使用 TLB1 映射 Linux 核心的虚拟地址空间)。与普通进程相比,核心进程访问存储器的效率要高一些。同时在 Linux 2.6 内核中,如果核心进程与普通进程的 static _ prio 参数相等时,核心进程将有机会获得更多的 CPU 资源。

基于以上原因,有许多用户使用 Linux 系统的核心进程实现自己的应用。但是用户在书写核心进程程序时,必须注意,核心进程属于 Linux 内核程序的一部分,是 Linux 系统信任的模块,核心进程的 Bug 可能引起整个 Linux 系统的崩溃。

Linux 系统中除 init _ task 进程外,其他进程都要使用 kernerl _ thread 函数创建。下文将以 核心进程 init 为例说明如何创建核心进程。核心进程 init 的源代码存放在 ./init/main.c 文件中,如下所示:

```
static int init(void * unused)
{
    lock_kernel();
    set_cpus_allowed(current, CPU_MASK_ALL);
...
}
```

在 rest _ init 函数中,Linux 系统使用 kernel _ thread 函数生成 init 进程。rest _ init 函数在 Linux 系统初始化时调用,其源代码在 ./init/main.c 文件中。

```
static void noinline rest_init(void)
    _releases(kernel_lock)
{
    kernel_thread(init, NULL, CLONE_FS | CLONE_SIGHAND);
    numa_default_policy();
...
    cpu_idle();
}
```

在 rest _ init 函数的开始将调用 kernel _ thread 函数创建 init 进程。kernel _ thread 函数共有三个输入参数 fn、arg 和 flags,该函数的源代码在 ./arch/powerpc/kernel/misc _ 32.S 文件中,在 kernel _ thread 函数中将使用系统调用 sys _ clone 来创建核心进程。kernel _ thread 函数的源代码实现较为简单,请读者自行阅读以了解其实现的细节。Linux 系统中还有许多核心进程如 keventd,kapmd,kswapd,pdflush 等,有些驱动程序也会根据需要创建核心进程。

5.2.2 普通进程

Linux 系统中存在两类用户进程，用户进程与用户线程。严格意义上说，只有拥有了进程描述符、PID、进程的正文段、用户进程的数据段和栈段、核心栈段的进程才被称为用户进程，而没有独立的数据段和栈段的进程被称为线程。

在 Linux 系统中，许多进程与其父进程共享同一段存储空间，这些进程严格说来还是线程；但是这些子进程可以通过 execve 系统调用创建其自己的进程空间，并与父进程分离，成为真正意义上的进程。

在 Linux 系统的早期版本中，使用系统调用 fork 创建子进程。此时子进程对父进程的数据段与栈段直接进行复制，这样做虽然保证了子进程的独立性，但是使用这种方法创建进程的开销过大。因此在 Linux 的后续版本中引入了线程的概念。此时，线程将与父进程共享数据段与栈段，从而极大地提高了操作系统创建进程的效率。在 Linux 系统中，进程和线程都有自己的进程描述符与核心栈段。其中，核心栈段用来支持用户进程或线程在 Linux 核心中执行。

Linux 系统使用 fork，clone 和 vfork 三个系统调用创建所有用户进程。

系统调用 fork 用来创建传统的用户进程，系统调用 clone 用来创建用户线程。系统调用 vfork 与 fork 系统调用功能类似。只是系统调用 vfork 将阻塞父进程的运行直到子进程执行完毕，或者执行 execve 系统调用替换从父进程继承的数据段与栈段。如果用户使用 fork 系统调用之后立即使用 execve 系统调用，可以使用系统调用 vfork 以提高效率。

在 Linux 系统中，系统调用 fork，clone 和 vfork 将分别调用 sys_fork，sys_vfork，sys_clone 函数。这些函数的源代码在 ./arch/powerpc/kernel/process.c 文件中：

```
int sys_clone(unsigned long clone_flags, unsigned long usp, int __user *parent_tidp,
      void __user *child_threadptr, int __user *child_tidp, int p6, struct pt_regs *regs)
{
  if (usp == 0) usp = regs->gpr[1]; /* stack pointer for child */
  return do_fork(clone_flags, usp, regs, 0, parent_tidp, child_tidp);
}

int sys_fork(unsigned long p1, unsigned long p2, unsigned long p3, unsigned long p4,
      unsigned long p5, unsigned long p6, struct pt_regs *regs)
{
  return do_fork(SIGCHLD, regs->gpr[1], regs, 0, NULL, NULL);
}

int sys_vfork(unsigned long p1, unsigned long p2, unsigned long p3, unsigned long p4,
      unsigned long p5, unsigned long p6, struct pt_regs *regs)
{
  return do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, regs->gpr[1],
      regs, 0, NULL, NULL);
}
```

以上三个函数最终都将调用 do_fork 函数进行进程创建。在 Linux 系统中，绝大多数进程都属于用户进程，如用户使用 Shell 界面执行的进程都是用户进程，所有的 Daemon 进程也都是用户进程。

5.3 Linux 系统中进程的状态转换

Linux 系统中，进程在执行过程中，可能会进行一系列的状态转换。这些状态包括 TASK_RUNNING，TASK_INTERRUPTIBLE，TASK_UNINTERRUPTIBLE，EXIT_DEAD 和 EXIT_ZOMBIE 等，进程的状态转换如图 5-6 所示。

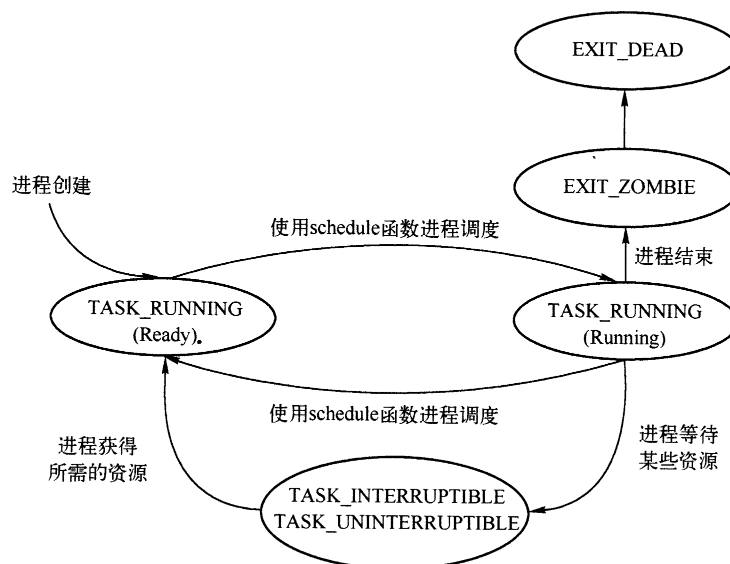


图 5-6 Linux 系统进程状态转换

Linux 系统的每个进程都有一个生命周期。都要经历创建、运行、结束等许多状态后结束运行。每个进程又可以创建子进程，这些子进程仍然要经历各自的生命周期。在 Linux 系统中，进程处于就绪态或者处于运行态时，其状态都为 TASK_RUNNING。

Linux 系统中的进程被创建后，将进入就绪态(Ready)等待执行，之后该进程等待合适的时机获得 CPU 资源，进入运行态(Running)。

进程在 CPU 中运行时，可能会由于等待某些信号或者中断事件而放弃在 CPU 中的运行，进入到 TASK_INTERRUPTIBLE 和 TASK_UNINTERRUPTIBLE，也可能被其他优先权较高的进程抢占或者使用当前进程的时间片而被调度程序切换到就绪态。

当进程执行完毕后，将放弃对 CPU 的使用，进入到结束态。Linux 系统中，一个进程在结束时，首先要进入 EXIT_ZOMBIE 状态，当父进程对此进程进行回收时，该进程将进入 EXIT_DEAD 状态，最后结束整个进程的运行。

一个进程在执行过程中，有可能会经历多次从运行到就绪，从就绪到运行，以及从运行到等待，从等待到就绪的状态转换。

5.3.1 进程的创建

在 Linux 系统中，程序员可以使用 kernel_thread，sys_fork，sys_clone 以及 sys_vfork 函数创建子进程。由 5.2.2 节所示，这些函数都将调用 do_fork 函数完成进程的创建，do_fork

函数的定义在./kernel/fork.c文件中。

do_fork函数一共有6个参数,如下所示:

(1) clone_flags。clone_flags参数是do_fork函数的重要参数,该参数由32位组成,每一位表示不同的含义,这些位的定义在./include/linux/sched.h文件中,clone_flags的主要参数如表5-1所示。

表5-1 clone_flags的主要参数

名　称	值	描　述
CLONE_VM	0x00000100	子进程与父进程共享存储空间
CLONE_FS	0x00000200	子进程与父进程共享使用的文件系统
CLONE_FILES	0x00000400	子进程与父进程共享打开的文件局柄
CLONE_SIGHAND	0x00000800	子进程与父进程共享信号量
CLONE_PTRACE	0x00002000	子进程可以被ptrace跟踪
CLONE_VFORK	0x00004000	使用vfork创建子进程时,此位置1
CLONE_PARENT	0x00008000	子进程的父进程为创建进程的父进程
CLONE_THREAD	0x00010000	子进程与父进程的线程组相同
CLONE_NEWNS	0x00020000	子进程与父进程不共享使用的文件系统
CLONE_SYSVSEM	0x00040000	子进程共享父进程的SEM_UNDO信号量
CLONE_UNTRACED	0x00800000	禁止对子进程的跟踪
CLONE_STOPPED	0x02000000	强制子进程进入TASK_STOP状态

(2) stack_start。该参数描述子进程的用户态堆栈指针。父进程将会为子进程创建新的用户堆栈。

(3) regs。该参数用来记录PowerPC处理器通用寄存器的值。

(4) stack_size。该参数没有被do_fork函数使用,恒为零。

(5) parent_tidptr和child_tidptr。这两个参数分别用来确定父进程与子进程的PID。

sys_fork、sys_vfork、sys_clone和kernel_thread调用do_fork函数时使用的参数不同。do_fork函数将根据这些不同的参数确定是创建核心进程,用户进程还是用户线程。

do_fork函数的源代码在./kernel/fork.c文件中。do_fork函数共由三部分组成:

(1) 使用alloc_pid函数创建进程的pid结构。

(2) 使用copy_process创建进程描述符,如果创建进程成功,则对子进程的状态进行检查。

(3) 调用wake_up_new_task函数将子进程加入到Linux系统的进程运行队列中,从而激活子进程。

1. 建立进程的pid结构

do_fork函数使用alloc_pid函数创建进程的pid结构,其源代码如下所示:

```
/* do_fork 函数源代码片段 1 */
long do_fork(unsigned long clone_flags, unsigned long stack_start,
         struct pt_regs *regs, unsigned long stack_size,
         int _user *parent_tidptr, int _user *child_tidptr)
```

```
{
    struct task_struct *p;
    int trace = 0;
    struct pid *pid = alloc_pid();
    long nr;

    if (! pid)
      return -EAGAIN;
    nr = pid->nr;
```

在这段程序中,do_fork 函数首先调用 alloc_pid 函数获得进程的 PID 号。alloc_pid 函数的源代码在 ./kernel/pid.c 文件中,该函数的源代码如下:

```
struct pid *alloc_pid(void)
{
  struct pid *pid;
  enum pid_type type;
  int nr = -1;

  pid = kmem_cache_alloc(pid_cachep, GFP_KERNEL);
  if (! pid)
    goto out;

  nr = alloc_pidmap(current->nsproxy->pid_ns);
  if (nr < 0)
    goto out_free;

  atomic_set(&pid->count, 1);
  pid->nr = nr;
  for (type = 0; type < PIDTYPE_MAX; ++type)
    INIT_HLIST_HEAD(&pid->tasks[type]);

...
out:
  return pid;
...
}
```

以上代码由以下几个步骤组成:

1) 首先使用 kmem_cache_alloc 函数从 Slab 分配器中获得子进程的 pid 结构使用的物理内存,Slab 分配器将在下文中详细讨论。

2) 调用 alloc_pidmap 函数分配进程的 ID 号。

3) 随后将子进程的 pid 结构的 count,nr 和 tasks 参数进行初始化。

alloc_pid 函数使用 FFB(Find First Bit)算法在全局变量 init_pspace 中查找未被使用的进程 ID 号。全局变量 init_pspace 的定义在 ./kernel/pid.c 文件中,该变量是一个 pspace 数据类型。数据结构 pspace 在 ./include/linux 目录的 pspace.h 文件中,其定义如下所示:

```
struct pspace {
    struct pidmap pidmap[PIDMAP_ENTRIES];
    int last_pid;
};

struct pidmap {
    atomic_t nr_free;
    void * page;
};
```

pspace 结构定义了 pidmap 参数和 last_pid 变量。其中，pidmap 变量用来存放进程 ID 号的位图信息，而 last_pid 变量用来存放在该位图中最后使用的 pid 号。

在 Linux 初始化时，init_space.pid_map[0 .. PIDMAP_ENTRIES] = { BITS_PER_PAGE, NULL }，即所有 pidmap 中的 nr_free 参数为 BITS_PER_PAGE，而 page 为空，表示在 init_pspace 变量中所有存放进程 ID 的位图都为空。

在 Linux 系统中，使用 alloc_pidmap 函数分配进程 ID 号，此函数的源代码在 ./kernel/pid.c 文件中。该函数将根据 FFB 算法遍历 &init_pspace→pidmap 中的空闲进程 ID 号，在遍历查找的过程中，分配 pidmap→page 所需的空间。

alloc_pidmap 函数的实现比较精巧，希望读者一定要真正理解这个函数的实现。本书不再拘泥于 alloc_pidmap 函数的实现细节。

2. 初始化子进程的进程描述符

```
/* do_fork 函数源代码片段 2 */
    p = copy_process(clone_flags, stack_start, regs, stack_size, parent_tidptr, child_tidptr,
    nr);
```

Linux 系统使用 copy_process 函数函数建立各类进程，该函数的源代码在 ./kernel/fork.c 文件中，下文将详细分析 copy_process 函数的主要源代码。

```
/* copy_process 函数源代码片段 1 */
static struct task_struct * copy_process(unsigned long clone_flags,
            unsigned long stack_start,
            struct pt_regs * regs,
            unsigned long stack_size,
            int _user * parent_tidptr,
            int _user * child_tidptr,
            int pid)
```

copy_process 函数中除了 pid 参数外，其他参数与 do_fork 函数的参数相同。从上文中的源代码中，可以发现 nr 变量由 alloc_pidmap 函数创建，即子进程的 ID 号。copy_process 函数的详细实现如下所示：

```
/* copy_process 函数源代码片段 2 */
{
  int retval;
```

```
struct task_struct *p = NULL;

if ((clone_flags & (CLONE_NEWNS|CLONE_FS)) == (CLONE_NEWNS|CLONE_FS))
  return ERR_PTR(-EINVAL);

if ((clone_flags & CLONE_THREAD) && !(clone_flags & CLONE_SIGHAND))
  return ERR_PTR(-EINVAL);

if ((clone_flags & CLONE_SIGHAND) && !(clone_flags & CLONE_VM))
  return ERR_PTR(-EINVAL);
```

copy_process 函数首先将对函数的入口参数进行检查，clone_flags 参数中的有些位具有相关性，有些位具有互斥性，如 CLONE_NEWNS 标志位和 CLONE_FS 不能同时为 1，CLONE_THREAD 标志位和 CLONE_SIGHAND 标志位必须同时为 1，CLONE_SIGHAND 标志位和 CLONE_VM 必须同时为 1。

```
/* copy_process 函数源代码片段 3 */
...
    p = dup_task_struct(current);
    if (!p)
      goto fork_out;

    rt_mutex_init_task(p);
...
    INIT_LIST_HEAD(&p->children);
    INIT_LIST_HEAD(&p->sibling);
    p->vfork_done = NULL;
    spin_lock_init(&p->alloc_lock);

    clear_tsk_thread_flag(p, TIF_SIGPENDING);
    init_sigpending(&p->pending);

    p->utime = cputime_zero;
    p->stime = cputime_zero;
    p->sched_time = 0;
    p->rchar = 0;      /* I/O counter: bytes read */
    p->wchar = 0;      /* I/O counter: bytes written */
    p->syscr = 0;      /* I/O counter: read syscalls */
    p->syscw = 0;      /* I/O counter: write syscalls */
```

随后，copy_process 函数调用 dup_task_struct 函数复制子进程的进程描述符。该函数的输入参数是 current，即当前运行进程的进程描述符。dup_task_struct 函数在 ./kernel/fork.c 文件中，其执行流程如下：

1）首先调用 prepare_to_copy 函数为复制子进程描述符作准备，该函数将对 PowerPC 处理器中的浮点、Altivec 和 SPE 寄存器与 current 描述符中的 thread 参数进行同步。

2）调用 alloc_task_struct 函数从 Slab 分配器中申请子进程描述符使用的物理内存。

3）调用 alloc _ thread _ info 函数从 Buddy System 中申请子进程描述符的 thread _ info 参数所使用的空间，该参数一共占用 8KB 空间。下文将详细介绍 Buddy System。

4）dup _ task 函数中使用 Gcc 提供的结构拷贝功能（struct copy）将父进程的进程描述符复制到子进程中，* tsk = * orig。

5）初始化子进程描述符的 state 和 thread _ info 参数。

6）调用 setup _ thread _ stack 函数将父进程描述符 thread _ info 参数中的内容复制到子进程描述符的 thread _ info 中。

7）初始化子进程描述符的其他参数。

dup _ task 函数执行完毕后，copy _ process 函数将对其返回状态进行判断。如果该函数成功返回，copy _ process 函数将对子进程描述符的其他属性进行初始化，如对 user、utime、io _ context、tgid 等一系列参数进行初始化，并将子进程与其他进程确立血缘关系。

```
/* copy _ process 函数源代码片段 4 */
...
    if ((retval = copy _ semundo(clone _ flags, p)))
      goto bad _ fork _ cleanup _ audit;
    if ((retval = copy _ files(clone _ flags, p)))
      goto bad _ fork _ cleanup _ semundo;
    if ((retval = copy _ fs(clone _ flags, p)))
      goto bad _ fork _ cleanup _ files;
    if ((retval = copy _ sighand(clone _ flags, p)))
      goto bad _ fork _ cleanup _ fs;
    if ((retval = copy _ signal(clone _ flags, p)))
      goto bad _ fork _ cleanup _ sighand;
    if ((retval = copy _ mm(clone _ flags, p)))
      goto bad _ fork _ cleanup _ signal;
    if ((retval = copy _ keys(clone _ flags, p)))
      goto bad _ fork _ cleanup _ mm;
    if ((retval = copy _ namespaces(clone _ flags, p)))
      goto bad _ fork _ cleanup _ keys;
    retval = copy _ thread(0, clone _ flags, stack _ start, stack _ size, p, regs);
    if (retval)
      goto bad _ fork _ cleanup _ namespaces;
    p->set _ child _ tid = (clone _ flags & CLONE _ CHILD _ SETTID)? child _ tidptr : NULL;
    /*
     * Clear TID on mm _ release()?
     */
    p->clear _ child _ tid = (clone _ flags & CLONE _ CHILD _ CLEARTID) ? child _ tidptr: NULL;
    p->robust _ list = NULL;
```

随后 copy _ process 函数调用 copy _ semundo，copy _ files，copy _ fs，copy _ sighand，copy _

signal, copy _ mm, copy _ keys, copy _ namespace 和 copy _ thread 函数对子进程描述符的其他属性进行赋值，在对这些属性进行赋值时将对 clone _ flags 参数进行检查以决定是否继承父进程的相应参数。

在以上程序中，copy _ mm 函数是一个非常重要的函数，该函数将创建子进程的进程地址空间，读者只有了解 Linux 系统内存管理的细节之后，才可能理解此函数。读者可以暂时这样理解 copy _ mm 函数的执行结果。当 clone _ flags 的 CLONE _ VM 位没有使能且父进程不是核心进程时，该函数将复制父进程内存地址空间，此时子进程与父进程的地址空间独立；如果父进程是核心进程时，子进程将使用 Linux 内核的地址空间；而当 clone _ flags 的 CLONE _ VM 位使能时，子进程将共享父进程的地址空间。

在以上程序中，copy _ thread 函数是另一个比较重要的函数。copy _ thread 函数初始化进程描述符的 thread 结构，包括进程的核心堆栈和寄存器信息，copy _ thread 函数的源代码在 ./arch/powerpc/kernel/process.c 文件中。

copy _ process 函数完成进程的复制工作后，将调用 sched _ fork 函数。

```
/* copy_process 函数源代码片段 5 */
...
    /* Perform scheduler related setup. Assign this task to a CPU. */
    sched_fork(p, clone_flags);

    /* Need tasklist lock for parent etc handling! */
    write_lock_irq(&tasklist_lock);

    /* for sys_ioprio_set(IOPRIO_WHO_PGRP) */
    p->ioprio = current->ioprio;

    /*
     * The task hasn't been attached yet, so its cpus_allowed mask will
     * not be changed, nor will its assigned CPU.
     *
     * The cpus_allowed mask of the parent may have changed after it was
     * copied first time - so re-copy it here, then check the child's CPU
     * to ensure it is on a valid CPU (and if not, just force it back to
     * parent's CPU). This avoids alot of nasty races.
     */
    p->cpus_allowed = current->cpus_allowed;
    if (unlikely(!cpu_isset(task_cpu(p), p->cpus_allowed) ||
            !cpu_online(task_cpu(p))))
        set_task_cpu(p, smp_processor_id());
...
} /* End copy_process */
```

sched _ fork 函数设置进程描述符与有调度有关的参数，该函数是 copy _ process 函数中调用的关键函数，其源代码在 ./kernel/sched.c 文件中，其主要源代码如下：

```
void fastcall sched_fork(struct task_struct *p, int clone_flags)
{
  int cpu = get_cpu();
#ifdef CONFIG_SMP
  cpu = sched_balance_self(cpu, SD_BALANCE_FORK);
#endif
  set_task_cpu(p, cpu);
  p->state = TASK_RUNNING;
  p->prio = current->normal_prio;
  INIT_LIST_HEAD(&p->run_list);
  p->array = NULL;
#ifdef CONFIG_PREEMPT
  /* Want to start with kernel preemption disabled. */
  task_thread_info(p)->preempt_count = 1;
#endif
...
  local_irq_disable();
  p->time_slice = (current->time_slice + 1) >> 1;
  p->first_time_slice = 1;
  current->time_slice >>= 1;
  p->timestamp = sched_clock();

  if (unlikely(! current->time_slice)) {
    current->time_slice = 1;
    task_running_tick(cpu_rq(cpu), current);
  }
  local_irq_enable();
  put_cpu();
}
```

sched_fork 函数的执行流程如下：

1）sched_fork 函数首先使用 get_cpu 函数获得当前进程使用的 CPU。get_cpu 函数调用 preempt_disable 函数，禁止对此 cpu 上的进程进行抢占，然后再调用 smp_processor_id 函数获得当前进程使用的 CPU。get_put 函数与 put_cpu 函数成对出现。

2）Linux SMP 将调用 sched_balance_self 函数，该函数的主要作用是根据 SMP 处理器中各个 CPU 的负载，选择目前相对最空闲的 CPU 运行子进程。Linux SMP 是一个相对复杂的系统，本书不对 Linux SMP 进行详细描述，在不远的将来我可能会专门写一本有关 Linux SMP 的书。

3）设置子进程的 state 参数为 TASK_RUNNING，prio 参数为当前进程的 normal_prio 参数。注意子进程将继承父进程的 normal_prio 参数，而不是 prio 参数。这样做的主要目的是为了避免因为 Priority Inherit 引起的 prio 参数与 normal_prio 参数不同步的问题。此外将

state 参数设置为 TASK _ RUNNING 并不意味着当前进程已经可以运行。只有子进程放入到进程运行队列后,才可能被 Linux 系统进程调度程序选中,获得 CPU 资源并运行。

4) 初始化子进程描述符的 run _ list,array 和 sched _ info 参数。如果 Linux 系统支持抢占式内核,此时将子进程描述符→thread _ info→preempt _ count 参数置为 1,表示子进程暂时不能被强占。

5) 根据当前进程的 time _ slice 参数,初始化子进程的 time _ slice 和 first _ time _ slice 参数,并改变当前进程的 time _ slice 参数。Linux 系统采用第 5.1.2 节中描述的算法调整子进程和当前进程的 time _ slice 和 first _ time _ slice 参数。

sched _ fork 函数执行完毕后,copy _ process 函数根据 clone _ flags 参数进一步对子进程描述符的其他属性进行初始化设置,直到子进程描述符创建完毕。

3. 激活子进程

copy _ process 函数执行完毕后,Linux 系统在 do _ fork 函数中使用 wake _ up _ new _ task 函数激活子进程,do _ fork 函数在激活子进程之前,使用"completion"机制对 vfork 系统调用进行专门的处理,其源代码如下:

```
/* do_fork 函数源代码片段 3 */
    if (! IS_ERR(p)) {
      struct completion vfork;

      if (clone_flags & CLONE_VFORK) {
        p->vfork_done = &vfork;
        init_completion(&vfork);
      }
...
      if (! (clone_flags & CLONE_STOPPED))
        wake_up_new_task(p, clone_flags);
      else
        p->state = TASK_STOPPED;
...
      if (clone_flags & CLONE_VFORK) {
        wait_for_completion(&vfork);
        if (unlikely (current->ptrace & PT_TRACE_VFORK_DONE)) {
          current->ptrace_message = nr;
          ptrace_notify ((PTRACE_EVENT_VFORK_DONE << 8) | SIGTRAP);
        }
      }
    } else {
      free_pid(pid);
      nr = PTR_ERR(p);
    }
    return nr;
} /* End do_fork */
```

在 Linux 系统中，“completion”是一种简单的同步机制。与 Linux 系统中的其他同步机制不同，complete 函数和 wait _ for _ completion 函数用于实现程序同步，而不是数据同步。

程序员在使用 completion 之前，必须在文件中包含 linux/completion.h，同时创建一个 struct completion 类型的变量。这个变量可以宏 DECLARE _ COMPLETION 静态地声明和初始化，也可以使用 init _ completion 函数动态的初始化 completion 变量。

如果一个程序需要等待某个过程的完成才能继续执行时，可以调用 wait _ for _ completion 函数等待 completion 事件的完成。如果该事件已经完成，则调用以下两个函数唤醒等待该事件的进程。

(1) void complete(struct completion * comp);

(2) void complete _ all(struct completion * comp);

complete 函数和 complete _ all 函数的区别在于前一个函数将只唤醒一个等待进程，而后一个函数唤醒等待该事件的所以进程。

在 Linux 系统中，complete 函数和 wait _ for _ completion 函数需要成对出现。complete 函数和 wait _ for _ completion 函数的源代码在 ./kernel/sched.c 文件中，本书不再对这两个函数进行详细说明。

在 do _ fork 函数中，使用了 init _ completion 和 wait _ for _ completion 函数处理 vfork 系统调用。这一组函数对父子进程进行同步处理，父进程需要等待子进程执行系统调用 execve 后或者执行完毕后，父进程才能继续执行。

在执行 execve 系统调用或者子进程执行完毕后，将会调用 mm _ release 函数。该函数调用 completion 函数标志当前的 vfork _ done 事件已经结束。该函数的源代码如下：

```
void mm _ release(struct task _ struct * tsk, struct mm _ struct * mm)
{
  struct completion * vfork _ done = tsk->vfork _ done;
...
  /* notify parent sleeping on vfork() */
  if (vfork _ done) {
    tsk->vfork _ done = NULL;
    complete(vfork _ done);
  }
...
} /* End mm _ release */
```

在 init _ completion 函数和 wait _ for _ completion 函数之间，do _ fork 函数才调用 wake _ up _ new _ task 函数激活子进程。因此如果 do _ fork 函数是在处理 vfork 系统调用，那么子进程一定在父进程之前执行。

wake _ up _ new _ task 函数的源代码在 ./kernel/sched.c 文件中，下文将详细介绍该函数。

wake _ up _ new _ task 函数一共有两个输入参数 p 和 clone _ flags，参数 p 指向子进程的进程描述符，而 clone _ flags 与 do _ fork 函数的 clone _ flags 参数的定义相同。

```
void fastcall wake_up_new_task(struct task_struct * p, unsigned long clone_flags)
{
  rq = task_rq_lock(p, &flags);
  BUG_ON(p->state != TASK_RUNNING);
  this_cpu = smp_processor_id();
  cpu = task_cpu(p);
  p->sleep_avg = JIFFIES_TO_NS(CURRENT_BONUS(p) *
    CHILD_PENALTY / 100 * MAX_SLEEP_AVG / MAX_BONUS);

  p->prio = effective_prio(p);
```

首先 wake_up_new_task 函数将进程运行队列加锁,然后获得 cpu 和 this_cpu 变量。其中 cpu 变量用来存放子进程将所使用 CPU 资源,而 this_cpu 变量存放当前进程使用的 CPU 资源,即父进程所使用的 CPU 资源。

这段函数初始化子进程描述符的 sleep_avg 参数,并调用 effective_prio 函数初始化其 prio 参数,effective_prio 函数的定义见 5.1.2 节。在 effective_prio 函数执行完毕后,wake_up_new_task 函数将使用以下程序将子进程加入到进程运行队列中:

```
if (likely(cpu == this_cpu)) {
  if (!(clone_flags & CLONE_VM)) {
    if (unlikely(!current->array))
      __activate_task(p, rq);
    else {
      p->prio = current->prio;
      p->normal_prio = current->normal_prio;
      list_add_tail(&p->run_list, &current->run_list);
      p->array = current->array;
      p->array->nr_active++;
      inc_nr_running(p, rq);
    }
    set_need_resched();
  } else
    __activate_task(p, rq);
  this_rq = rq;
```

以上这段程序是 wake_up_new_task 函数的关键部分。这段程序使用__activate_task 函数将子进程加入到进程运行队列中,然后等待调度程序使子进程获得 CPU 资源。其详细执行步骤如下:

1) 比较 cpu 与 this_cpu 变量是否相等。在单 CPU 处理器系统中,cpu 与 this_cpu 变量一定相等;在多处理器系统中,如 SMP 系统, cpu 与 this_cpu 变量在大多数情况下也相等,但是有时 cpu 和 this_cpu 的值并不相等。

2) 如果 clone_flags 参数中的 CLONE_VM 位没有使能,即父子进程没有共享虚存空间,此时 Linux 系统需要子进程优先运行。此时,子进程的 run_list 参数被加入到当前进程的 run

_ list 参数之前,即在 Linux 系统的进程运行队列中,子进程在父进程之前。采用这种方式创建子进程的主要原因是 Linux 系统采用了 COW(Copy-On-Write)技术。当父子进程没有共享虚存空间时,子进程率先执行,从而有效地降低因为处理 COW 页而产生的数据异常的频率,在 Linux PowerPC 中,处理这个异常的函数为 do _ page _ fault 函数,该函数的详细描述见 7.5.2 节。

(3) 如果 clone _ flags 参数中的 CLONE _ VM 位使能,即父子进程共享虚存空间时,wake _ up _ new _ task 函数使用_ activate _ task 函数将子进程加入到进程运行队列中,此时子进程将加入到父进程之后(注意,使用 list _ add _ tail 函数将子进程加入到进程运行队列的父进程之前)。

```
    } else {
        this_rq = cpu_rq(this_cpu);
        p->timestamp = (p->timestamp - this_rq->timestamp_last_tick)
                        + rq->timestamp_last_tick;
        __activate_task(p, rq);
        if (TASK_PREEMPTS_CURR(p, rq))
            resched_task(rq->curr);
        task_rq_unlock(rq, &flags);
        this_rq = task_rq_lock(current, &flags);
    }
    task_rq_unlock(this_rq, &flags);
} /* End wake_up_new_task */
```

如果 cpu 与 this _ cpu 变量不相等,则表示父子进程没有在相同的 CPU 中执行,此时必须对子进程的 timestamp 参数重新进行计算。在 Linux SMP 系统中,每一个 CPU 都有自己的进程运行队列,并有各自的 timestamp _ last _ tick 参数。

timestamp _ last _ tick 参数存放最近一次系统时钟异常发生的时间,而在 Linux SMP 中,每一个 CPU 都有独立的系统时钟异常。因此当父子进程在不同的 CPU 中执行时,必须进行时间同步。

wake _ up _ new _ task 函数最后将调用 task _ rq _ unlock 将进程运行队列解锁。do _ fork 函数在执行完 wake _ up _ new _ task 函数后将子进程加入到进程运行队列中,此后子进程可以被进程调度程序选中,获得 CPU 资源运行。

4. 进程地址空间的更换

在 Linux 系统中,核心进程和用户进程调用 do _ execve 函数实现进程地址空间的更换,这一过程也被称为可执行文件的执行。do _ execve 函数从文件系统中获得可执行文件的映像,然后将当前进程的地址空间更改为可执行文件中存放的地址空间。

核心进程调用 do _ execve 函数更换进程地址空间后,将当前核心进程使用的正文段、数据段更换为 do _ execve 函数指定的可执行文件中的正文段和数据段,并为该进程重新建立用户堆栈空间,将核心进程更改为用户进程。目前 Linux 系统中只有核心进程 init 需要调用 do _ execve 函数将核心进程改变为用户进程。核心进程一旦改变为用户进程后,将不能恢复为核心进程。

用户进程调用 do _ execve 函数更换进程地址空间的时,将当前用户进程使用的正文段、数据段更换为 do _ execve 函数指定的可执行文件中的正文段和数据段。do _ execve 函数在 ./fs/exec.c 文件中。do _ execve 函数的实现,关键在于 search _ binary _ handler 函数。在此函数中判断当前可执行文件的类型,然后使用不同的加载程序将存放在文件系统中的可执行文件加载到内存中,并替换当前进程的地址空间,读者可以参考图 2-1 了解有关 do _ execve 函数的执行流程。

在 Linux 系统中,最常用的可执行文件类型为 ELF 格式,在 do _ execve 函数中进程可以使用 load _ elf _ binary 函数加载可执行文件,load _ elf _ binary 函数的定义在 ./fs/binfmt _ elf.c 文件中。该函数的主要作用是根据存放在文件系统中的可执行文件对当前进程的内存描述符,即 current→mm 参数进行重新赋值,并使用 elf _ map 和 do _ mmap 函数将可执行文件中的正文段和数据段映射到内存中,最后 load _ elf _ binary 函数将调用 start _ thread 函数将进程描述符中存放的通用寄存器进行初始化。理解该函数的关键在于理解 ELF 文件的格式,对此本书将不再进行详细说明。

5.3.2 进程的结束

在 Linux 系统中,有些进程始终无法结束。如初始化进程 init _ task,init 进程和许多核心进程,然而绝大多数用户进程可以使用 exit 系统调用结束。

在 Linux 系统中,仅使用 exit 系统调用无法将进程及其进程使用的资源完全释放。一个进程的结束必须要其父进程协调完成,其中子进程使用 exit 系统调用释放子进程使用的大部分资源,而父进程使用系统调用 wait4 释放子进程使用的进程描述符。

1. exit 系统调用

在 Linux 内核中,系统调用 exit 使用 sys _ exit 函数实现,该函数在 ./kernel/exit.c 文件中,其源代码如下:

```
asmlinkage long sys_exit(int error_code)
{
  do_exit((error_code&0xff)<<8);
}
```

Linux 系统调用 sys _ exit→do _ exit 函数实现 exit 系统调用,do _ exit 函数的源代码在 ./kernel/exit.c 文件中。

```
fastcall NORET_TYPE void do_exit(long code)
```

do _ exit 函数只有一个输入参数 code,该参数表示进程的返回状态,该函数没有返回值。此外在 void 参数之前,do _ exit 函数有一个前缀 NORET _ TYPE。NORET _ TYPE 前缀在 ./include/linux/linkage.h 文件中定义,其等效于“/* */”。因此对于编译器而言,这个前缀并没有什么实际意义,Linux 源代码设置 NORET _ TYPE 前缀,用来表示当前函数不但没有返回值,而且也不会返回。

当一个进程执行完毕 do _ exit 函数后,Linux 系统将释放该进程使用的所有系统资源。因此 do _ exit 函数不会返回到 sys _ exit 函数中,sys _ exit 函数也无法结束当前系统调用,回到该进程地址空间继续执行。do _ exit 函数使用 schedule 函数将对当前进程进程进行切换,因此

对于当前进程 do_exit 函数将不会返回。在 Linux 进程管理与调度模块中有许多函数都无法返回到调用者。do_exit 函数首先进行一些状态检查，其代码如下：

```
{
...
if (unlikely(in_interrupt()))
  panic("Aiee, killing interrupt handler!");

if (unlikely(! tsk->pid))
  panic("Attempted to kill the idle task!");

if (unlikely(tsk == child_reaper(tsk))) {
  if (tsk->nsproxy->pid_ns != &init_pid_ns)
    tsk->nsproxy->pid_ns->child_reaper = init_pid_ns.child_reaper;
  else
    panic("Attempted to kill init!");
}

if (unlikely(current->ptrace & PT_TRACE_EXIT)) {
  current->ptrace_message = code;
  ptrace_notify((PTRACE_EVENT_EXIT << 8) | SIGTRAP);
}
```

这段代码的详细说明如下：

(1) 如果 do_exit 函数由中断服务程序调用，则调用 panic 函数将系统挂起。因为在正常情况下，中断服务程序不可能调用 do_exit 函数。Linux 系统使用 in_interrupt 函数判断当前运行的程序是否为中断服务程序。

(2) 如果当前进程描述符的 pid 参数为 0，则当前进程为 init_task 进程，此时将调用 panic 函数，将系统挂起。因为 Linux 系统不能将 init_task 进程结束。

(3) 如果当前进程是 child_reaper 进程，此时将调用 panic 函数，将系统挂起。child_reaper 进程用来处理所有没有父进程的子进程。在 Linux 系统中，有时子进程在结束之前，父进程已经结束，此时 Linux 系统使用 child_reaper 进程处理这些子进程。child_reaper 的进程描述符被 ./init/main.c 文件的 init 函数赋值，该进程与 init 进程等效。

(4) 如果当前子进程被 ptrace 跟踪，Linux 系统使用信号机制通知 ptrace 进程。当前正在被跟踪的进程即将被中止。

do_exit 函数完成参数检查后，将继续执行。

```
if (unlikely(tsk->flags & PF_EXITING)) {
  printk(KERN_ALERT "Fixing recursive fault but reboot is needed! \n");
  if (tsk->io_context)
    exit_io_context();
  set_current_state(TASK_UNINTERRUPTIBLE);
  schedule();
}
tsk->flags |= PF_EXITING;
```

如果当前进程描述符的 flags 参数的 PF _ EXITING 位为 1,表示当前进程执行的 do _ exit 函数是被 do _ exit 函数调用的,即在 do _ exit 函数中递归调用 do _ exit 函数,因为进程描述符的 flags 参数的 PF _ EXITING 位只可能在 do _ exit 函数中被设置。

正常情况下不可能出现执行上述源代码的情况。这里提醒读者注意,在一个并不过分追求效率的程序里,不妨对可能出现的条件进行检查,也许这些小的处理将有助于诊断出一个异常复杂的系统级 bug。

随后这段程序将当前进程描述符的 flags 参数的 PF _ EXITING 位置 1,之后 do _ exit 函数继续执行以下程序,将当前进程使用的各项资源逐一释放。

```
...
    exit_mm(tsk);

    if (group_dead)
        acct_process();
    exit_sem(tsk);
    _exit_files(tsk);
    _exit_fs(tsk);
    exit_thread();
    cpuset_exit(tsk);
    exit_keys(tsk);
...
    exit_notify(tsk);
```

Linux 系统在进程结束时需要释放其占用的所有资源。在 5.3.1 节中可以发现,子进程在创建时从父进程继承了一系列资源。这些资源必须在进程结束时被释放。进程释放资源的过程基本上是进程创建时获得相应资源的的逆过程。只是在子进程释放资源时,需要调用 exit _ notify 函数。

彻底理解 exit _ notify 函数需要读者对 Linux 进程间通信机制有一定认识。exit _ notify 函数使用 do _ notify _ parent 函数向父进程发出结束信号,之后父进程选择合适的时机运行,释放子进程的进程描述符。进程描述符中有两个参数,parent 和 real _ parent 参数,用来描述进程的父进程,这两个参数在大多数时间是相同的,只有使用 ptrace 系统调用对当前进程进行跟踪时,这两个参数不同。子进程结束时,将向该进程的 parent(parent 进程是指 parent 参数指向的进程)进程发出结束信号。

在某些情况下,parent 进程可能会提前结束,此时 parent 进程需要选择其他进程托管其子进程。一般来说,parent 进程在其结束前使用当前进程组的第一个进程托管其子进程。如果该进程组的第一个进程也已经结束,将使用 init 进程,即 Linux 系统的进程 1,托管子进程,这也是 Linux 系统的 init 进程也被称为 child _ reaper 进程的原因。

exit _ notify 函数调用 do _ notify _ parent 函数后,将子进程描述符的 state 参数改为 EXIT _ ZOMBIE。如图 5-6 所示,当进程进入 EXIT _ ZOMBIE 状态后,将再也没有机会被 schedule 函数选中获得 CPU 资源。

```
        if (unlikely(! list_empty(&tsk->pi_state_list)))
          exit_pi_state_list(tsk);
        if (unlikely(current->pi_state_cache))
          kfree(current->pi_state_cache);
        /*
         * Make sure we are holding no locks:
         */
        debug_check_no_locks_held(tsk);

        if (tsk->io_context)
          exit_io_context();

        if (tsk->splice_pipe)
          _free_pipe_info(tsk->splice_pipe);

        preempt_disable();
        /* causes final put_task_struct in finish_task_switch(). */
        tsk->state = TASK_DEAD;

        schedule();
      ...
      } /* End do_exit */
```

do_exit 函数随后将继续释放子进程的所有资源，然后将子进程描述符的 state 参数设置为 TASK_DEAD，最后调用 schedule 函数进行切换，从而最终完成子进程的运行。此时子进程将完全释放所使用的资源，只有其进程描述符仍然存在。

Linux 系统在处理进程的结束时设置了两个状态 EXIT_ZOMBIE 和 TASK_DEAD。其原因是为了防止在进程在结束时受到进程调度程序的干扰。因此 Linux 系统首先将进程状态设置为 EXIT_ZOMBIE，此时子进程已经不能被调度程序选中，exit_notify 函数执行后，父进程可能被唤醒，此时父进程可以安全地将子进程描述符释放。最后 do_exit 函数将进程状态设置为 TASK_DEAD，调用 schedule 函数将子进程切换出去，至此子进程结束其生命周期，完成运行的全过程。

2. waitpid 系统调用

waitpid 系统调用的主要作用是释放子进程残留的进程描述符，从而最终释放子进程占用的系统空间。Linux 系统调用 sys_waitpid→sys_wait4→do_wait 函数实现 wait 系统调用。在 Linux 系统中，父进程使用系统调用 wait4 和 waitpid 释放子进程的进程描述符。一个典型的父进程等待子进程结束的例子如下所示：

```
      main()
      {
      ...
        pid = fork();
        if(pid = = 0) {
      ...
          /* child process */
```

```
    exit(0);
  }
...
  waitpid(pid,NULL,0)
}
```

在上述程序中,子进程调用 exit 函数向父进程发出信号,表示子进程已经执行完毕。如果程序员在书写程序时,忘记在子进程结束时调用 exit 函数也没有关系,因为 Linux 系统在子进程结束后,会自动调用这个函数,即便程序员没有显式地编写这个函数。

子进程结束后,会发出信号唤醒父进程,此时父进程将调用 waitpid 函数将子进程描述符释放。在以上程序中,如果子进程不结束,父进程将一直在 waitpid 函数中等待子进程的结束。如果在上述程序中父进程没有调用 waitpid 函数就退出执行,则其子进程将成为僵尸进程,其子进程的进程描述符将由 init 进程释放。

do_wait 函数的主要作用是释放子进程的进程描述符,该函数在 ./kernel/exit.c 文件中,其主要代码如下所示:

```
static long do_wait(pid_t pid, int options, struct siginfo_user * infop,
        int_user * stat_addr, struct rusage_user * ru)
{
  DECLARE_WAITQUEUE(wait, current);
  struct task_struct * tsk;
  int flag, retval;

  add_wait_queue(&current->signal->wait_chldexit,&wait);
repeat:
  flag = 0;
  current->state = TASK_INTERRUPTIBLE;
  read_lock(&tasklist_lock);
  tsk = current;
  do {
...
      list_for_each(_p,&tsk->children) {
        p = list_entry(_p, struct task_struct, sibling);
        ret = eligible_child(pid, options, p);
        if (! ret)
          continue;
        switch (p->state) {
...
        default:
          if (p->exit_state == EXIT_DEAD)
            continue;
          if (p->exit_state == EXIT_ZOMBIE) {
```

```
            if (ret == 2)
              goto check_continued;
            if (! likely(options & WEXITED))
              continue;
            retval = wait_task_zombie(p, (options & WNOWAIT), infop, stat_addr, ru);
            if (retval != 0)
              goto end;
            break;
          }
check_continued:
          flag = 1;
          if (! unlikely(options & WCONTINUED)) continue;
          retval = wait_task_continued(p, (options & WNOWAIT), infop, stat_addr, ru);
          if (retval != 0) goto end;
          break;
        } /* End switch p->state */
      }; /* End list_for_each */
...
      tsk = next_thread(tsk);
      BUG_ON(tsk->signal != current->signal);
    } while (tsk != current);
...
      if (flag) {
        retval = 0;
        if (options & WNOHANG)
          goto end;
        retval = -ERESTARTSYS;
        if (signal_pending(current))
          goto end;
        schedule();
        goto repeat;
      }
        retval = -ECHILD;
      end:
...
    return retval;
  } /* End do_wait */
```

do_wait 函数的主体在标号 repeat 和 end 之间。如果父进程没有收到子进程的结束信号,则将在 schedule 函数执行后睡眠,直到子进程发出结束信号,唤醒父进程。当父进程被激活后,将进行以下操作:

1) 设置当前进程描述符的 state 参数为 TASK_INTERRUPTIBLE,获得锁 tasklist_lock

为即将执行的代码加锁。然后将 tsk 变量设置为 current. tsk 变量是 do-while 循环使用的临时变量，该值 next _ th 被 read 函数不断进行更新，直到父进程将所有子进程扫描完毕。

2）在 do-while 循环中，父进程对所有子进程进行检查，如果子进程状态为 EXIT _ DEAD 或 EXIT _ ZOMBIE 时，将调用 wait _ task _ continued→put _ task _ struct 函数释放子进程描述符。

5.3.3 进程的等待

进程在运行过程中，有时需要等待某些信号或者中断。此时，Linux 系统为了提高 CPU 的使用率，会主动地将当前进程切换到等待状态，让出 CPU 资源，由调度程序选择合适的进程使用 CPU。当进程需要等待某些信号或者中断资源时，该进程将进入等待状态；而当进程需要的信号或者中断来临时将在处于等待状态的进程唤醒。

Linux 系统为了有效地对处于等待状态的进程进行管理，设置了等待队列 WQ(Wait Queue, WQ)及对等待队列进行管理的一系列函数。当进程因为某种原因进入等待状态后，Linux 系统会将这些进程插入到相应的等待队列中；当进程退出等待状态时，这些进程将从相应的等待队列中摘除。

1. 等待队列(Wait Queue)

为了便于对处于等待态的进程进行管理和控制，Linux 系统可以根据需要，设置许多等待队列。Linux 系统使用数据结构 wait _ queue _ head _ t 描述这些等待队列。

```
struct _ wait _ queue _ head {
  spinlock _ t lock;
  struct list _ head task _ list;
};

typedef struct _ wait _ queue _ head wait _ queue _ head _ t;
```

数据结构 wait _ queue _ head _ t 有两个参数，分别是 lock 和 task _ list。其中 lock 参数保存访问等待队列的自旋锁，而 task _ list 参数保存在这个队列中存放的进程描述符链表，即在该等待队列中等待的进程。

Linux 系统使用 init _ waitqueue _ head 函数初始化等待队列。系统程序员可以在文件系统、网络协议栈及初始化一些设备驱动程序时调用该函数，以初始化这些模块使用的等待队列。程序员可以根据需要在一个模块中设置多条等待队列，init _ waitqueue _ head 函数首先初始化该队列使用的自旋锁，然后将队列头指针进行初始化。

该函数的源代码在 ./kernel/wait.c 文件中，其源代码详解如下：

```
void init _ waitqueue _ head(wait _ queue _ head _ t * q)
{
  spin _ lock _ init(&q->lock);
  INIT _ LIST _ HEAD(&q->task _ list);
}
```

在 Linux 系统中，处于等待状态的进程与在在等待队列中的一个 Entry 相对应。每一个等待队列中的 Entry 使用 wait _ queue _ t 结构进行描述，wait _ queue _ t 结构如下所示：

```
typedef struct _ wait _ queue wait _ queue _ t;

struct _ wait _ queue {
  unsigned int flags;
#define WQ _ FLAG _ EXCLUSIVE   0x01
  void * private;
  wait _ queue _ func _ t func;
  struct list _ head task _ list;
};
```

wait _ queue 结构较为简单，一共定义了四个参数 flags，private，func 和 task _ list 参数。

(1) private 参数保存进程描述符。

(2) func 参数保存等待队列中的唤醒函数指针。在等待队列中，每一个 Entry 可以独立设置各自的唤醒函数。

(3) task _ list 参数将所有属于同一个等待队列的进程链接在一起。

(4) flags 参数相对较为复杂。该参数表示当前等待队列中 Entry 的属性。这个属性的值可以是 0 也可以是宏 WQ _ FLAG _ EXCLUSIVE。当此属性为 0 时，当前 Entry 对应的进程与其他进程共享等待条件；当此属性为 WQ _ FLAG _ EXCLUSIVE，即为 1 时，当前 Entry 对应的进程独享等待条件，即当此进程被激活后将独占当前资源而导致其他使用这一资源的进程进入等待状态。

Linux 系统使用 add _ wait _ queue 函数将进程加入到等待队列的头部。该函数的源代码如下所示。add _ wait _ queue 函数将进程加入到等待队列的头部的同时将等待队列中相应 Entry 的 flags 位置为 0。

```
void fastcall add _ wait _ queue(wait _ queue _ head _ t * q, wait _ queue _ t * wait)
{
  unsigned long flags;

  wait->flags &= ~WQ _ FLAG _ EXCLUSIVE;
  spin _ lock _ irqsave(&q->lock, flags);
  _ add _ wait _ queue(q, wait);
  spin _ unlock _ irqrestore(&q->lock, flags);
}
```

Linux 系统还可以使用 add _ wait _ queue _ exclusive 函数将进程加入到等待队列的尾部。该函数的源代码如下所示。add _ wait _ queue 函数将进程加入到等待队列的尾部的同时将等待队列中相应 Entry 的 flags 位置为 WQ _ FLAG _ EXCLUSIVE。

```
void fastcall add _ wait _ queue _ exclusive(wait _ queue _ head _ t * q, wait _ queue _ t * wait)
{
  unsigned long flags;

  wait->flags |= WQ _ FLAG _ EXCLUSIVE;
  spin _ lock _ irqsave(&q->lock, flags);
  _ add _ wait _ queue _ tail(q, wait);
  spin _ unlock _ irqrestore(&q->lock, flags);
}
```

由以上源代码我们可以发现，在一个进程的等待队列 WQ 中，flags 位为 0 的 Entry 被存放在等待对列的头部，而 flags 位为 WQ_FLAG_EXCLUSIVE 的 Entry 将被存放在等待队列的尾部。

如果一个程序使用等待队列的 task_list→next 访问等待队列时可以访问到 flags 参数为 0 的 Entry，而使用等待队列的 task_list→prev 访问等待队列时可以访问到 flags 参数为 1 的 Entry。Linux 系统按照 flags 位的不同将在等待队列的进程分为两类的主要目的是，加快等待队列中进程的移出速度。

在 Linux 系统中，一般只将一个 flags 位为 WQ_FLAG_EXCLUSIVE 的进程从等待队列中移出。因为如果两个 flags 位为 WQ_FLAG_EXCLUSIVE 的进程被同时激活时，只有一个进程可以获得这个资源，另一个没有获得此资源的进程仍然需要被放入等待队列中，从而降低了 CPU 的使用效率。

add_wait_queue 和 add_wait_queue_exclusive 函数使用了 spin_lock_irqsave 函数以防止进程操作等待队列进行时被其他进程干扰，因为其他进程也有可能会操作此等待队列。

2. 进程运行状态到等待状态的迁移

在 Linux 系统中，一个进程无法获得某些资源，如 Mutex、Semaphore、信号或者某些中断时，将进入等待状态。一个进程也可以根据需要主动选择进入等待状态。比如进行 DMA 操作时，一个进程将与 DMA 操作相关的相应寄存器设置完毕后，就可以主动进入等待状态，等待 DMA 完成中断，并由中断服务程序激活该等待进程。

程序员可以根据需要使用 wait_event，wait_event_timeout，wait_event_interruptible 和 wait_event_interruptible_timeout 函数自动将进程从运行状态迁移到等待状态。

wait_event 和 wait_event_timeout 函数将进程放入等待队列中，并将当前进程的状态设置为 TASK_UNINTERRUPTIBLE，即在等待队列中的进程不可以被信号激活，而只能由中断事件激活。而 wait_event_interruptible 和 wait_event_interruptible_timeout 函数将进程放入等待队列中，并将当前进程的状态设置为 TASK_INTERRUPTIBLE，即在等待队列中的进程可以被信号和中断事件激活。

wait_event_interruptible_timeout 和 wait_event_timeout 函数会为当前等待进程设置一个定时器。当等待进程在指定的时间内没有被信号或者中断激活时，这个定时器将激活等待进程。

以上这四个函数功能及实现类似，下文将以 wait_event 函数为例说明这一类函数。在 Linux 系统中，wait_event 函数使用宏定义的方式实现，宏 wait_event 的源代码如下所示：

```
#define wait_event(wq, condition)            \
do {                                         \
  if (condition)                             \
    break;                                   \
  _wait_event(wq, condition);                \
} while (0)
```

宏 wait_event 有两个参数 wq 和 condition。宏 wait_event 首先对 condition 进行判断，如果条件为真，则退出 do_while 循环，表示当前进程不需要进入等待状态，否则继续调用_wait_event 函数。_wait_event 函数是宏 wait_event 的主体，其输入参数与宏 wait_event 完全相

同,其实现如下:

```
#define _ wait _ event(wq, condition)                                          \
do {                                                                            \
  DEFINE _ WAIT(_ wait);                                                        \
                                                                                \
  for (;;) {                                                                    \
    prepare _ to _ wait(&wq, & _ wait, TASK _ UNINTERRUPTIBLE);                 \
    if (condition)                                                              \
      break;                                                                    \
    schedule();                                                                 \
  }                                                                             \
  finish _ wait(&wq, & _ wait);                                                 \
} while (0)
```

宏_ wait _ event 的执行流程如下:

1) 首先使用宏 DEFILE _ WAIT 定义一个 wait _ queue _ t 类型的临时变量_ wait,并将此变量的 func,private 和 task _ list 参数分别初始化为 autoremove _ wake _ function,current 和 LIST _ HEAD _ INIT((name).task _ list)。

2) 然后宏_ wait _ event 调用 prepare _ to _ wait 函数为进程进入等待队列 wq 作准备。

3) prepare _ to _ wait 函数执行完毕后,_ wait _ event 宏需要对 condition 条件再次判断。

4) 如果此时 condition 为真。即_ wait _ event 函数在执行宏 DEFINE _ WAIT 和 prepare _ to _ wait 过程中,Linux 系统中的 condition 已经被改写为真,此时进程不会再次被放入等待队列,而是直接跳出当前 for 循环,之后调用 finish _ wait 函数。如果此时 condition 为假,宏_ wait _ event 调用 schedule 函数切换当前进程。当等待进程被激活并重新获得 CPU 运行后,将运行宏_ wait _ event 中 schedule 函数之后的程序,即继续调用 prepare _ to _ wait,直到 condition 条件为真后退出 for 循环。

5) 退出 for 循环后,宏_ wait _ event 调用 finish _ wait 函数将当前进程的状态设置为 TASK _ RUNNING,然后将_ wait 变量从 wq 等待队列中摘除。

prepare _ to _ wait 函数是宏_ wait _ event 中的关键函数,其源代码如下:

```
void fastcall
prepare _ to _ wait(wait _ queue _ head _ t *q, wait _ queue _ t *wait, int state)
{
  unsigned long flags;

  wait->flags &= ~WQ _ FLAG _ EXCLUSIVE;
  spin _ lock _ irqsave(&q->lock, flags);
  if (list _ empty(&wait->task _ list))
    _ add _ wait _ queue(q, wait);
  /*
   * don't alter the task state if this is just going to
   * queue an async wait queue callback
   */
```

```
        if (is_sync_wait(wait))
            set_current_state(state);
        spin_unlock_irqrestore(&q->lock, flags);
}
```

prepare_to_wait 函数首先将_wait 变量的 flags 参数设置为 0，然后初始化等待队列 wq。prepare_to_wait 函数使用自旋锁将该函数执行的程序保护起来，同时保证在函数的执行过程中禁止外部中断。

如果当前进程使用同步 I/O，进程的状态被设置为 TASK_UNINTERRUPTIBLE；如果使用异步 I/O 操作，则不改变进程的状态。Linux 系统经常使用同步 I/O 模型，即当进程发出 I/O 请求命令，如使用 read 函数对 I/O 设备进行读操作时，read 函数将被阻塞，执行 read 函数的进程将进入等待状态，直到 I/O 请求被满足后 read 函数才能够返回。

使用异步 I/O 模型的进程，发出 I/O 请求命令后，如使用 read 函数对 I/O 设备进行读操作时，read 函数将会立即返回，当前进程会继续执行。在该进程执行的某个阶段使用 select 系统调用，检测需要获取的 I/O 资源是否在系统中有效，之后才能够使用 read 函数获得的数据。

在 Linux 系统中，除了使用上述的四个函数可以将进程从运行状态切换到等待状态之外，进程对临界数据访问也可能会使其进入等待状态。Linux 系统使用互斥锁 Mutex，信号灯 Semaphore 和自旋锁三种方式对临界数据进行保护。

互斥锁 Mutex 和信号灯 Semaphore 机制也是采用了将进程加入等待队列的方法。Mutex 和 Semaphore 使用的等待队列，其 Entry 的 flags 参数为 1，因为对 Mutex 和 Semaphore 资源的访问是互斥的。

请读者参考 ./include/asm-powerpc/semaphore.h 文件获得 Mutex 和 Semaphore 的详细代码。这部分源代码较为简单，其中 Mutex 是 Semaphore 的一个特例，信号量为 1 的 Semaphore 等效于 Mutex。在 Linux 系统中使用函数 up 和 down 对 Semaphore 中的信号量进行操作，从而实现进程的状态转换。

提醒读者注意，在 Linux PowePC 中，自旋锁的实现机制与 Mutex 和 Semaphore 完全不同。自旋锁的详细实现见 4.1.1 节。

3. 进程等待状态到就绪状态的迁移

当进程被激活后，必须首先进入就绪状态后，才能被 schedule 函数选中获得 CPU 资源。在 Linux 系统中，进程不能从等待状态直接迁移到运行状态。Linux 系统主要使用以下宏将进程从等待状态迁移到就绪状态。

- 宏 wake_up。该宏将激活所有在等待队列中 flags 为 0 的进程(即在等待队列中共享等待条件的进程)，和一个 flags 状态为 1(flag 状态为 WQ_FLAG_EXCLUSIVE)的进程(即一个在等待队列中独占等待条件的进程)。在 Linux 系统中，该宏的使用频率较高。
- 宏 wake_up_nr。该宏将激活所有 flags 为 0 的进程，和多个 flags 为 1 的进程。在实际应用中，很少将多个 flags 位为 1 的进程激活。
- 宏 wake_up_all。该宏将唤醒在等待队列中的所有进程。
- 宏 wake_up_interruptible，wake_up_interruptible_nr 和 wake_up_interruptible_all 与以上的三个宏一一对应，只是这组宏只能唤醒状态为 TASK_INTERRUPTIBLE 的进程，而以上的三个宏可以唤醒状态为 TASK_UNINTERRUPTIBLE 和 TASK_INTERRUPTIBLE 的进程。

在 Linux 系统中，将进程唤醒的宏需要与使进程进入等待状态的宏（如 wait_event）成对使用。这些宏的源代码在 ./include/linux/wait.h 文件中，如下所示：

```
#define wake_up(x)                                    \
  __wake_up(x, TASK_UNINTERRUPTIBLE | TASK_INTERRUPTIBLE, 1, NULL)
#define wake_up_nr(x, nr)                             \
  __wake_up(x, TASK_UNINTERRUPTIBLE | TASK_INTERRUPTIBLE, nr, NULL)
#define wake_up_all(x)                                \
  __wake_up(x, TASK_UNINTERRUPTIBLE | TASK_INTERRUPTIBLE, 0, NULL)
#define wake_up_interruptible(x)                      \
  __wake_up(x, TASK_INTERRUPTIBLE, 1, NULL)
#define wake_up_interruptible_nr(x, nr)               \
  __wake_up(x, TASK_INTERRUPTIBLE, nr, NULL)
#define wake_up_interruptible_all(x)                  \
  __wake_up(x, TASK_INTERRUPTIBLE, 0, NULL)
#define wake_up_locked(x)                             \
  __wake_up_locked((x), TASK_UNINTERRUPTIBLE | TASK_INTERRUPTIBLE)
#define wake_up_interruptible_sync(x)                 \
  __wake_up_sync((x),TASK_INTERRUPTIBLE, 1)
```

在这些宏中，宏 wake_up 和 wake_up_interruptible 最为常用。宏 wake_up_interruptible 和 wake_up 的区别在于调用__wake_up 函数时传递的参数不同。

宏 wake_up 将 TASK_UNINTERRUPTIBLE | TASK_INTERRUPTIBLE 的值传递给__wake_up 函数的 mode 参数，而宏 wake_up_interruptible 将 TASK_INTERRUPTIBLE 的值传递给__wake_up 函数。__wake_up 函数的源代码在 ./kernel/sched.c 文件中：

```
void fastcall __wake_up(wait_queue_head_t *q, unsigned int mode, int nr_exclusive, void *key)
{
  unsigned long flags;

  spin_lock_irqsave(&q->lock, flags);
  __wake_up_common(q, mode, nr_exclusive, 0, key);
  spin_unlock_irqrestore(&q->lock, flags);
}
```

__wake_up 函数共有 3 个参数。

- 参数 q 用来指向等待队列。
- nr_exclusive 用来存放一共需要激活多少个独占等待条件的进程。
- 参数 key 没有使用。

该函数调用__wake_up_common 函数唤醒相应的进程，__wake_up_common 函数可以操作进程的等待队列，因此需要使用自旋锁保护这段代码。__wake_up_common 函数调用 try_to_wake_up 函数把在等待队列中的进程激活。

在 Linux 系统中，除了宏 wake_up 和 wake_up_interruptible 之外，还有一些函数可以将处于等待状态的进程激活，如 5.3.1 节提到的 wake_up_new_task 函数可以将新进程激活；wake_up_process 函数可以将指定的进程激活；wake_up_state 函数可以将处于指定状态的

进程激活。这些函数最终会调用 try _ to _ wake _ up 函数将进程激活,如图 5-7 所示。

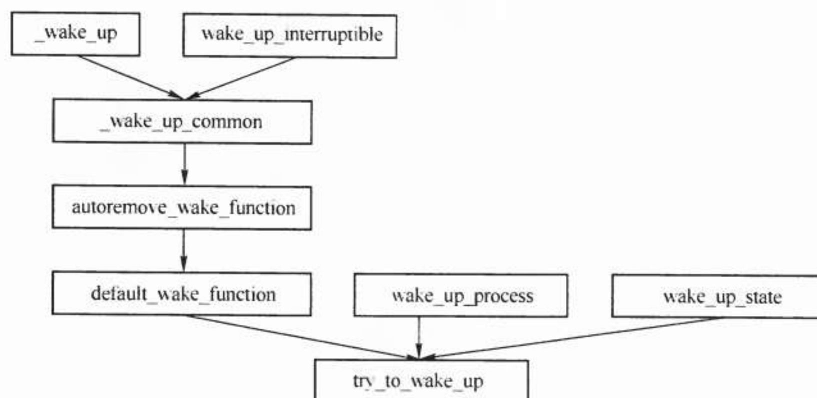


图 5-7　wake _ up 类函数的调用关系

在 Linux 系统中,绝大多数 wake _ up 类函数最终调用 try _ wake _ up 函数唤醒在等待队列中的进程,该函数的源代码在 ./kernel/sched.c 文件中。

```
static int try_to_wake_up(struct task_struct *p, unsigned int state, int sync)
{
```

try _ to _ wake _ up 函数一共有 3 个参数。该函数的主要作用是激活在等待队列中状态与 state 参数相同的进程。

- 参数 p 指向即将从等待队列中激活进程的进程描述符。
- 参数 state 存放 try _ to _ wake _ up 函数的访问标示。
- 参数 sync 确定当前这次激活操作是否为同步方式。

```
...
  rq = task_rq_lock(p, &flags);
  old_state = p->state;
  if (! (old_state & state))
    goto out;
...
```

这段函数首先使用 task _ rq _ lock 函数获得进程 p 所属的进程运行队列 rq,进程运行队列的详细描述见下文。在 Linux 系统中,将等待进程激活,等效于将进程从等待队列中搬移到进程的运行队列。

如果进程 p 的 state 参数与 try _ to _ wake _ up 函数的 state 参数不相等则跳转到 out 处退出函数的执行。这段程序使用了大段的源代码对 Linux SMP 进行处理,本书不再介绍这段代码。

```
if (old_state == TASK_UNINTERRUPTIBLE) {
  rq->nr_uninterruptible--;
  p->sleep_type = SLEEP_NONINTERACTIVE;
} else
  if (old_state & TASK_NONINTERACTIVE)
    p->sleep_type = SLEEP_NONINTERACTIVE;
```

这段程序的主要作用是识别一个进程的交互性。在 Linux 系统中，交互式进程不是创建时确立的，而是在进程运行过程中动态确立的。当进程被重新激活时，将有机会调整进程的交互性。

这段程序首先分析进程的状态，如果进程从 TASK_UNINTERRUPTIBLE 状态中进入就绪状态，则将运行队列的 nr_uninterruptible 参数减 1，然后将进程的睡眠状态赋值为 SLEEP_NONINTERACTIVE。如果进程的 TASK_NONINTERACTIVE 状态位有效，进程的睡眠状态被赋值为 SLEEP_NONINTERACTIVE。nr_uninterruptible 参数与进程的睡眠状态将在下文详细介绍。

```
    activate_task(p, rq, cpu == this_cpu);
...
out_running:
    p->state = TASK_RUNNING;
out:
    task_rq_unlock(rq, &flags);
    return success;
}
```

最后，try_to_wake_up 函数使用 active_task 函数将进程加入到运行队列中，将进程的状态改写为 TASK_RUNNING，激活该进程。

5.4 进程的调度

上文介绍了图 5-6 中，一个进程从创建到就绪，从运行到结束，从进程运行到等待，从等待到运行，一共四种状态的迁移。本节将详细介绍进程从就绪到运行、从运行到就绪的状态迁移，即进程的调度。

Linux 系统支持多任务与多处理器，其中多个进程可以在多个 CPU 中并发执行。在单处理器中，每一个进程在同一时刻内只有一个可以获得 CPU 资源。Linux 系统的调度程序根据进程的特点在合适的时机选择最合适的进程获得 CPU 资源。为支持多个进程，Linux 将 CPU 的时间划分为若干个时间片，多个进程可以轮流使用这些时间片，从而多个进程可以共享一个或多个 CPU 资源。

在 Linux 系统中，进程调度是一个关键模块。进程调度的速度也是衡量操作系统性能的重要指标。在 Linux 系统的进程调度中，需要在功能与性能之间进行权衡，进程调度对操作系统的总体设计有着决定性影响。

Linux 系统是一个分时操作系统，基于时间片进行调度。一个时间片的大小由每秒钟 CPU 执行系统时钟异常的频率决定，在两个系统时钟异常之间的时间被称作一个时间片。系统时钟异常的频率越高，Linux 系统的响应速度越快，但是 CPU 用于处理时钟中断的时间就越多，用于处理进程执行的时间越短。操作系统可以根据处理器及实际应用的需求，合理设置系统时钟异常的频率。在 Linux 系统中，宏 HZ 描述系统时钟异常的频率，读者可以参考 5.1.2 节了解 HZ 的详细含义。

Linux 系统中的进程分为实时进程与普通进程，这两类进程采用不同的调度策略。而普通进

程又分为交互式进程、非交互式进程与批处理进程,这几种进程也采用不同的调度策略。这些不同类型的进程给 Linux 的进程调度子系统设计增添了一定难度。Linux 系统的进程调度需要对交互式进程、批处理进程、普通进程和实时进程采用不同的策略以保证这些进程的特殊需要。

- 交互式进程一般与用户的输入密切相关,如 vi 进程和 shell 进程。这些进程因为等待用户的输入而处于休眠状态。但是 Linux 系统发现有用户输入信息时,必须保证这类进程能够及时响应。
- 批处理进程对响应速度没有要求,但是这类进程需要获得较高的"平均执行时间"。
- 实时进程需要及时处理所获得信息。在 Linux 系统中,这类进程需要最高的响应速度。Linux 必须保证实时进程在规定的时间内处理完毕相应的信息。

在合理处理这几类的进程的同时,进程调度必须保证 CPU 的时间相对公平地分配给所有的进程。在理论界,进程调度的研究一直是操作系统的重中之重。Linux 2.6 中采用了 O(1)调度策略调度系统中所有的进程。

Linux 系统为实现其调度策略设置了许多数据结构,其中最重要的数据结构是进程运行队列。进程运行队列中存放了许多与进程调度有关的数据结构。Linux 系统中,进程运行队列用来存放所有处于就绪状态的进程。与 Linux 进程调度有关的源代码在 ./kernel/sched.c 和 ./kernel/sched.h 文件中。

5.4.1 进程运行队列

Linux 系统为每一个处理器提供一条进程运行队列(RQ,Run Queue)。在单处理器系统中,进程运行队列的个数为 1;在多处理器系统中,如 SMP 系统,进程运行队列 RQ 的个数与 SMP 处理器中 CPU 的个数相同。Linux 系统使用数据结构 rq 描述进程运行队列。为简单起见,下文使用 RQ 称呼 Linux 系统中的进程运行队列。

RQ 中包含了许多与 Linux SMP 有关的参数如 cpu_load,active_balance 等,Linux 系统使用这些参数实现进程在不同 CPU 之间的迁移,即一个进程可以根据 CPU 的负载度选择在哪一个 CPU 上执行,也可以根据当前进程的亲和属性决定当前进程优先使用哪个 CPU。用户可以使用 sys_sched_setaffinity 系统调用确定进程的亲和属性。

在 Linux SMP 系统中,进程调度的主要设计目标是在不失调度算法公平与高效的基础上追求负载均衡。对于 SMP 系统,负载均衡也意味着高效。在本书中将不会对 Linux SMP 进行过多讨论。Linux SMP 系统基于单 CPU 的 Linux 系统。如果读者真正掌握了基于单 CPU 的 Linux 系统和一个可以用来实现 SMP 构架的处理器,那么理解 Linux SMP 系统是水到渠成的。

1. RQ 中的重要参数

Linux 系统使用 rq 结构描述进行运行队列,该结构的定义在 ./kernel/sched.c 文件中。在 rq 结构中除了与 Linux SMP 有关的参数外,还有一些其他的重要参数。

```
struct rq {
    spinlock_t lock;
    unsigned long nr_running;
    unsigned long raw_weighted_load;
    unsigned long long nr_switches;
    unsigned long nr_uninterruptible;
```

```
    unsigned long expired_timestamp;
    unsigned long long most_recent_timestamp;
    struct task_struct *curr, *idle;
    unsigned long next_balance;
    struct mm_struct *prev_mm;
    struct prio_array *active, *expired, arrays[2];
    int best_expired_prio;
    atomic_t nr_iowait;
}
```

- lock 参数。RQ 属于 Linux 系统的临界数据。当一个进程访问 RQ 访时，需要使用自旋锁对 RQ 进行保护。
- nr_running 参数。该参数记录在 RQ 中处于 TASK_RUNNING 状态的进程个数。
- raw_weighted_load。该参数表示当前 RQ 的负载度，等于在 RQ 中的所有进程的负载度之和。进程描述符的 load_weight 参数用来描述每个进程的负载度。
- nr_switches 记录处理器一共进行了多少次进程切换操作。
- nr_uninterruptible 记录在 RQ 中一共有多少个处于 TASK_UNINTERRUPTIBLE 状态的进程。
- expired_timestamp。该参数存放在 RQ 中 expired 队列的等待时间，当 RQ 的 expired 队列与 active 队列进行交换时，该参数被初始化为 0。在 scheduler_tick 函数中，该参数被重新进行赋值。
- most_recent_timestamp。该参数记载当前 RQ 最近一次发生调度的时间，或者最近一次产生系统时钟异常的时间。在 Linux 系统进行进程切换或者系统时钟异常时，该参数被 update_cpu_clock 函数赋值。
- curr,idle。这两个参数分别保存当前 CPU 正在运行的 current 进程和 idle 进程的描述符。
- prev_mm。当发生进程调度时，被切换进程的 active_mm 参数保存在 RQ 的 prev_mm 参数中。
- best_expired_prio。该参数存放在 RQ 的 expired 队列中优先级最高的进程描述符的 prio 参数。
- nr_iowait 参数。该参数保存在 RQ 中正在等待磁盘 I/O 操作结束的进程数量。

在 rq 结构中，核心参数是 active，expired 与 array[2]。其中 active 与 expired 参数是 prio_array 类型的指针，array 是 prio_array 类型的数组。prio_array 数据结构的定义如下：

```
struct prio_array {
    unsigned int nr_active;
    DECLARE_BITMAP(bitmap, MAX_PRIO+1); /* include 1 bit for delimiter */
    struct list_head queue[MAX_PRIO];
};

#define DECLARE_BITMAP(name,bits) \
    unsigned long name[BITS_TO_LONGS(bits)]
```

其参数详解如下：

- nr _ active 参数。该参数记录 prio _ array 中的进程数量。
- bitmap 参数。该参数表示在当前 prio _ array 中有哪些 queue 是有效的。bitmap 参数由 5 个无符号长整型组成。在 Linux 系统中，MAX _ PRIO 的值为 140，因此在 bitmap 数组的 160 位中，只有前 140 位有效并与参数 queue 一一对应。当 bitmap 的第一位为 1 时表示 queue[0]中具有有效数据，当 bitmap 的第二位为 1 时表示 queue[1]中具有有效数据，依此类推。
- queue。该参数指向进程描述符链表。其中 queue [0] 用来保存所有优先权为 0 的进程描述符链表，而 queue [1] 用来保存所有优先权为 1 的进程描述符链表。

在进程运行队列中，active，expired 与 arrays[2]参数的关系如图 5-8 所示。

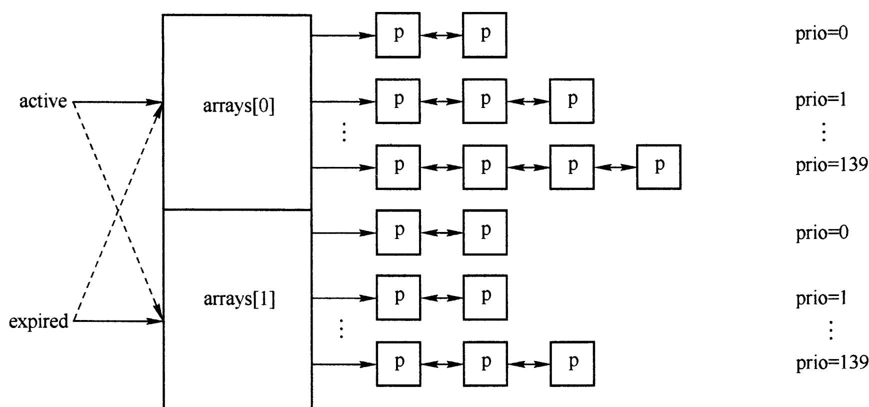


图 5-8 Linux 进程运行队列的结构图

在 Linux 系统初始化时，active 和 expired 指针分别指向 arrays[0]和 arrays[1]。在 Linux 系统中，active 指向当前活跃进程对列，而 expired 指向将时间片使用完毕的进程对列。在 Linux 系统中，调度程序通过操作进程运行队列，实现进程调度。

在 Linux 系统中，active 队列中优先权最高的进程将获得 CPU 资源运行。当进程将其时间片使用完毕后，该进程描述符将从 active 队列中移到 expired 队列中。当 active 队列中没有进程描述符时，active 指针将与 expired 指针互换。

2. 对 RQ 进行操作的函数

Linux 系统使用 dequeue _ task，enqueue _ task ，enqueue _ task _ head 和 requeue _ task 函数对 RQ 进行操作。这 4 个函数都有两个输入参数，分别是 p 指针和 arrary 指针，指针 p 指向进程描述符，array 指针指向 RQ→active 队列或者 RQ→exipred 队列，这 4 个函数的简单描述如下：

- dequeue _ task 函数将 p 指向的进程描述符从 array 队列中摘除。
- enqueue _ task 函数将 p 指向的进程描述符加入到 array 队列的尾部。
- enqueue _ task _ head 函数将 p 指向的进程描述符加入到 array 队列的头部。
- requeue _ task 函数将 p 指向的进程描述符从 array 队列中摘除，然后再加入到 array 队列的尾部。

其中，enqueue _ task 函数的源代码如下所示：

```
static void enqueue_task(struct task_struct *p, struct prio_array *array)
{
    sched_info_queued(p);
    list_add_tail(&p->run_list, array->queue + p->prio);
    __set_bit(p->prio, array->bitmap);
    array->nr_active++;
    p->array = array;
}
```

enqueue_task 函数将进程加入到 array 队列的尾部，然后更新 array->bitmap 和 nr_active 参数，最后将进程描述符的 array 参数指向 RQ 的 active 指针。enqueue_task_head 函数的实现机制与 enqueue_task 函数类似。而 dequeue_task 函数是 enqueue_task 函数的逆过程，而其源代码如下所示：

```
static void dequeue_task(struct task_struct *p, struct prio_array *array)
{
    array->nr_active--;
    list_del(&p->run_list);
    if (list_empty(array->queue + p->prio))
        __clear_bit(p->prio, array->bitmap);
}
```

dequeue_task 函数首先将 RQ 中的 nr_active 参数减 1，然后将进程从 RQ 中删除。如果在 RQ 中，不存在优先权和当前进程优先权相同的进程描述符链表，则将 array->bitmap 中的对应位清除。

5.4.2 系统时钟异常

在 Linux 系统中，一个进程只能以时间片为单位使用 CPU，当这个时间片耗尽后，系统将剥夺该进程的 CPU 使用权，把 CPU 分配给其他的进程使用，如果在系统中没有比当前进程优先权更高的进程，该进程将重新获得 CPU 资源，继续执行。在 Linux 系统中，许多与进程调度有关的系统参数随进程运行时间的长短而发生变化，这些变化的系统参数协调着 Linux 系统的正常运转。

Linux 系统使用系统时钟异常将 CPU 的运行时间分解成为一个个时间片。系统时钟异常处理程序将调整当前进程与调度有关的参数，从而激活整个 Linux 系统的运转。系统时钟异常程序在 Linux 中处于核心位置，因此系统时钟异常也被称为 Linux 系统的“心跳”。

PowerPC 处理器提供一个 32 位的计数器 DEC(Decrementer Register)支持系统时钟异常。Linux PowerPC 在初始化时为 DEC 寄存器赋予初值，之后 DEC 寄存器将自减。当 DEC 的值从 1 到 0 进行跳变时产生 Decrementer 异常。在 Linux PowerPC 中，该异常被用作系统时钟异常。系统时钟异常处理程序还会重新对 DEC 寄存器赋值，从而 Linux PowerPC 周而复始地产生系统时钟异常。

基于 E500 内核的 Linux PowerPC 使用 timer_interrupt 函数处理 Decrementer 异常，并在 timer_interrupt 函数中调用 scheduler_tick 函数进一步处理系统时钟异常。

1. timer _ interrupt 函数

Decrementer 异常在 ./arch/powerpc/kernel/head _ booke.h 文件中初始化，其源代码如下：

```
#define DECREMENTER_EXCEPTION                                          \
  START_EXCEPTION(Decrementer)                                         \
  NORMAL_EXCEPTION_PROLOG;                                             \
  lis   r0,TSR_DIS@h;       /* Setup the DEC interrupt mask */         \
  mtspr SPRN_TSR,r0;   /* Clear the DEC interrupt */                   \
  addi r3,r1,STACK_FRAME_OVERHEAD;                                     \
  EXC_XFER_LITE(0x0900, timer_interrupt)
```

通过以上代码发现，PowerPC 处理器发生 Decrementer 异常时，将调用 timer _ interrupt 函数。timer _ interrupt 函数的源代码在 ./arch/powerpc/kernel/time.c 文件中。该函数的源代码如下：

```
void timer_interrupt(struct pt_regs * regs)
{
  struct pt_regs *old_regs;
  int next_dec;
  int cpu = smp_processor_id();
  unsigned long ticks;
  u64 tb_next_jiffy;
```

timer _ interrupt 函数只有一个输入参数 regs，regs 参数在宏 EXC _ XFER _ LITE 中初始化，该参数保存 E500 内核的系统寄存器，有关宏 EXC _ XFER _ LITE 的详细说明请参考 6.4.2 节。在 timer _ interrupt 函数中，使用 smp _ processor _ id 函数获得当前进程使用的 CPU 号，对于单 CPU 系统，该值为 0。

```
#ifdef CONFIG_PPC32
  if (atomic_read(&ppc_n_lost_interrupts) != 0)
    do_IRQ(regs);
#endif

  old_regs = set_irq_regs(regs);
  irq_enter();
```

这段程序的执行顺序如下：

1) 在进入系统时钟异常程序时，如果发现还有未处理的外部中断则调用 do _ IRQ 函数处理这个外部中断，do _ IRQ 函数是外部中断处理函数，该函数将在第 6 章详细介绍。

2) 使用 set _ irq _ regs 函数将 pt _ regs 结构指针存入 Linux 系统为每一个 CPU 准备的 per _ cpu _ irq _ reg 结构中，同时将之前存放在 per _ cpu _ irq _ reg 结构中的数据备份到 old _ regs 变量中。pt _ regs 结构存放 PowerPC 处理器中的系统寄存器。

3) 调用 irq _ enter 函数将当前进程的 thread -> preempt _ count 参数与 HARDIRQ _ OFFSET 相加，表示 Linux 系统已经在中断处理程序的上下文中运行。

Linux SMP 需要为每一个 CPU 准备一套 per _ cpu 结构，并使用宏 DEFINE _ PER _ CPU 定义 per _ cpu 结构中的参数。timer _ interrupt 函数使用 per _ cpu _ irq _ reg 结构的主要原因是在 SMP 处理器中，每一个 CPU 都会处理各自的 Decrementer 异常，并会调用 timer _ interrupt 函数。因此在此函数中，必须使用 per _ cpu _ irq _ reg 结构保存来自不同 CPU 的系统寄存器。

```
profile_tick(CPU_PROFILING);
calculate_steal_time();
```

calculate _ steal _ time 调整在 timer _ interrupt 函数中进程的运行时间，传统的做法是将在此期间的进程的运行时间统一用 Jiffy 来计算。但是此函数的运行有可能被 CPU 中的其他异常所中断。

在 PowerPC 处理器中，Decrementer 异常的优先权最低，因此别的异常很有可能会中断 timer _ interrupt 函数的运行，从而导致当前进程在计算运行时间时产生误差。

calculate _ steal _ time 函数通过读取 PowerPC 处理器中的 TB 寄存器对进程的运行时间进行计算，从而可以消除这一误差。对此函数有兴趣的读者可以参考以下 URL：

http://ozlabs.org/pipermail/linuxppc64-dev/2006-February/008169.html

```
while ((ticks = tb_ticks_since(per_cpu(last_jiffy, cpu))) >= tb_ticks_per_jiffy) {
  per_cpu(last_jiffy, cpu) += tb_ticks_per_jiffy;
```

随后 timer _ interrupt 函数判断从上一次 CPU 进入该函数到现在为止，所使用的时间是否超过 tb _ ticks _ per _ jiffy。如果超过，将执行 while 循环。在一般情况下该 while 循环只执行一次，但是在某些特殊情况下，如在一段时间内外部中断异常频繁，Linux 系统没有机会进入到 timer _ interrupt 函数，此时 while 循环将运行多次。

tb _ ticks _ per _ jiffy 变量在 ./arch/powerpc/kernel/time.c 文件中定义，该变量用来表示在一个 Jiffy 中有多少个 ticks。Linux PowerPC 在初始化时将为此变量赋值。该变量的计算公式如下：

```
tb_ticks_per_jiffy = ppc_tb_freq / HZ;
```

ppc _ tb _ freq 表示在 PowerPC 处理器中 TB 寄存器使用的时钟频率。在基于 E500 内核的 PowerPC 处理器中，该频率为 CPU 主频的 1/8，当然用户也可以为 TB 提供外部时钟。对于一个 1GHz 主频的 PowerPC 处理器，如果 HZ 的值为 1000 且没有为 TB 提供外部时钟，则 tb _ ticks _ per _ jiffy 为 125000，即在 1 个 Jiffy 中含有 125000 个 ticks。在上述这段程序使用 tb _ ticks _ per _ jiffy 变量更新 per _ cpu _ last _ jiffy 变量后，继续执行以下程序：

```
...
  if (!cpu_is_offline(cpu))
    account_process_time(regs);
```

如果当前 CPU 没有进入 offline 状态，则调用 account _ process _ time 函数。从 Linux 系统将一个 CPU 标记为 offline 状态到这个 CPU 真正进入 offline 状态，需要一段时间，在这段时间里 CPU 仍然有可能收到 Decrementer 异常时，并调用 timer _ interrupt 函数，在这种情况下，time _ interrupt 函数将不调用 account _ process _ time 函数。

在 account_process_time 函数中将调用 scheduler_tick 函数，scheduler_tick 函数是系统时钟异常处理的核心。

```
        if (cpu != boot_cpuid)
            continue;
        write_seqlock(&xtime_lock);
        tb_next_jiffy = tb_last_jiffy + tb_ticks_per_jiffy;
        if (per_cpu(last_jiffy, cpu) >= tb_next_jiffy) {
            tb_last_jiffy = tb_next_jiffy;
            do_timer(1);
            timer_recalc_offset(tb_last_jiffy);
            timer_check_rtc();
        }
        write_sequnlock(&xtime_lock);
    } /* end while */
```

如果执行 timer_interrupt 函数的 CPU 不是 Boot CPU 则返回 while 循环的开始处，判断是否应该退出 while 循环，否则 Boot CPU 将作一些处理后再返回到 while 循环。在 Linux SMP 中，共有两类 CPU，分别是 BSP(Boot Strap Processor)和 AP(Application Processor)。其中 BSP 用来引导 Linux SMP 系统，而 AP 用作运算。Linux SMP 系统尽管追求完全的负载均衡，但是 Boot CPU 还是不可避免地比其他 CPU 多做一部分工作。

```
        next_dec = tb_ticks_per_jiffy - ticks;
        set_dec(next_dec);

        irq_exit();
        set_irq_regs(old_regs);
    } /* End timer_interrupt */
```

这段程序首先计算 next_dec，然后将该值重新写入 PowerPC 处理器中的 DEC 寄存器中，以确定下一次 Decrementer 异常来临的时间。在 timer_interrupt 函数中，while 循环体中的程序运行时间并不固定，有时运行时间较长，有时运行时间较短，因此 timer_interrupt 函数每次都需要重新计算 next_dec，以校对时间间隔。在 timer_interrupt 函数的最后，将调用 irq_exit 函数退出中断状态，调用 set_irq_regs 函数恢复 per_cpu_irq_reg 结构。

2. scheduler_tick 函数

scheduler_tick 函数在 timer_interrupt 函数中调用，该函数是系统时钟异常处理程序的核心。scheduler_tick 函数对当前正在执行进程的许多参数进行调整，并决定当前进程是否应该被切换。

```
    void scheduler_tick(void)
    {
        unsigned long long now = sched_clock();
        struct task_struct *p = current;
        int cpu = smp_processor_id();
```

```
    struct rq * rq = cpu_rq(cpu);

    update_cpu_clock(p, rq, now);

    if (p == rq->idle)
      /* Task on the idle queue */
      wake_priority_sleeper(rq);
    else
      task_running_tick(rq, p);

  } /* End scheduler_tick */
```

scheduler_tick 函数没有输入参数，也没有返回值。该函数的执行流程如下：

1) 使用 sched_clock 函数获得当前系统时间 now，当前进程描述符的 current 指针和当前进程使用的 CPU，获得当前进程所在的进程运行队列 RQ。

2) scheduler_tick 首先调用 update_cpu_clock 函数同步当前进程的 sched_time 参数，然后将 last_run 参数和 RQ 的 most_recent_timestamp 同步。Linux 系统在 scheduler_tick 函数中调用 update_cpu_clock 函数的作用是可以更加精确地计算出当前进程使用的 CPU 时间。这里的处理方法是与 Linux 2.6.19 内核相对而言的。

熟悉 Linux 2.6.19 内核的读者，可以发现 Linux 2.6.20 内核使用 most_recent_timestamp 参数替代了 RQ 中的 timestamp_last_tick 参数。由于许多读者并不熟悉 Linux 2.6.19 内核进程调度的这段源代码，本书不再详细说明采用这种方法的优点。对此有兴趣的读者可以浏览以下 URL：

http://lkml.org/lkml/2006/11/17/305

3) 如果当前进程为 idle 进程，则调用 wake_priority_sleeper 函数。如果当前处理器不支持 SMT 结构，wake_priority_sleeper 函数将为一个空函数。目前 Freescale 的 PowerPC 处理器都没有支持 SMT 结构。

如果当前处理器支持 SMT 结构，即便当前进程为 idle 进程也不一定表示当前 CPU 中没有其他活跃的进程，因为在 SMT 处理器中可能还有一些在硬件上睡眠的进程。因此 wake_priority_sleeper 函数需要检查在 RQ 中活跃的进程数目，即 rq→nr_running 是否为 0。当 RQ 中活跃进程的数目不为 0 表示当前 SMT 处理器中还有在硬件上睡眠的进程。此时使用 resched_task 函数将该进程的 flags 参数设置为 TIF_NEED_RESCHED，然后退出 scheduler_tick 函数。

如果当前进程不是 idle 进程，scheduler_tick 函数调用 task_running_tick 函数继续进行处理。task_running_tick 函数返回后，scheduler_tick 函数也将执行完毕。task_running_tick 函数一共有两个参数，参数 rq 指向当前进程运行队列 RQ，而参数 p 指向当前进程使用的描述符 current。

3. task_running_tick 函数

task_running_tick 函数在 scheduler_tick 函数中调用。该函数一共有两个参数 rq 和 p，rq 指向进程运行队列，而 p 指向当前进程的描述符，其源代码如下：

```
/* task_running_tick 函数源代码片段 1 */
static void task_running_tick(struct rq *rq, struct task_struct *p)
{
  if (p->array != rq->active) {
    /* Task has expired but was not scheduled yet */
    set_tsk_need_resched(p);
    return;
  }
```

task_running_tick 函数首先进行参数检查，如果当前进程不在 RQ 的 active 队列中，则使用 set_tsk_need_resched 函数将当前进程状态改变为 TIF_NEED_RESCHED 以便重新进程调度，并退出 task_running_tick 函数。Linux 系统中很少出现这种情况。

```
/* task_running_tick 函数源代码片段 2 */
    spin_lock(&rq->lock);
    if (rt_task(p)) {
      /*
       * RR tasks need a special form of timeslice management.
       * FIFO tasks have no timeslices.
       */
      if ((p->policy == SCHED_RR) && !--p->time_slice) {
        p->time_slice = task_timeslice(p);
        p->first_time_slice = 0;
        set_tsk_need_resched(p);

        /* put it at the end of the queue: */
        requeue_task(p, rq->active);
      }
      goto out_unlock;
    }
```

scheduler_tick 函数将对临界资源 RQ 进行操作，因此在本段程序使用 spin_lock 函数将 RQ 加锁。

如果当前进程是实时进程，scheduler_timer 函数需要做一些专门的处理。在 Linux 系统中，实时进程可以采用两种策略进行调度，SCHED_RR 和 SCHED_FIFO。SCHED_RR 策略使用基于优先权的时间片轮转策略，优先级相同的进程轮流使用 CPU 资源；SCHED_FIFO 策略采用先来先服务的方法，只有优先权较高的进程执行完毕后，其他优先权较低的进程和与当前优先权相同的实时进程才可以执行。

如果当前实时进程采用 SCHED_FIFO 策略，则直接跳转到 out_unlock 标号处，完成 task_running_tick 函数的执行。采用 SCHED_FIFO 策略的实时进程，其 time_slice 参数没有意义，该类进程只有执行完毕或者有更高优先权的进程时，才会让出 CPU 资源。

如果当前实时进程采用 SCHED_RR 调度策略，则进一步判断其时间片 time_slice 是否已经耗尽。如果 time_slice 没有耗尽则直接跳转到 out_unlock 标号处，完成 scheduler_tick

函数的执行。如果 time_slice 已经耗尽,则按以下步骤进行处理:

1) 使用 task_timeslice 函数重新分配当前进程的时间片 time_slice,该函数的详细说明见5.1.2节。

2) 将当前进程的 first_time_slice 参数置为0。

3) 使用 set_tsk_need_resched 函数将此进程的 flags 状态设置为 TIF_NEED_RESCHED。当系统时钟异常返回时,该进程将被切换。

4) 使用 requeue_task 函数将当前进程从 RQ 相应的 active 队列中摘除,然后再加到 RQ 的 active 队列的队尾,该函数的说明见5.4.1节。

以上程序中简单介绍了 task_running_tick 函数对实时进程的处理。可以发现,Linux 系统对实时进程的处理较为简单。在 Linux 系统中,SCHED_RR 和 SCHED_FIFO 两种调度算法的实现也较为简单。

下文将介绍 task_running_tick 函数对普通进程的处理。在 Linux 系统中,进程调度子系统对普通进程的处理较为复杂,普通进程采用 SCHED_NORMAL 或者 SCHED_BATCH 策略进行调度。

```
/* task_running_tick 函数源代码片段 3 */
    if (! --p->time_slice) {
      dequeue_task(p, rq->active);
      set_tsk_need_resched(p);
      p->prio = effective_prio(p);
      p->time_slice = task_timeslice(p);
      p->first_time_slice = 0;

      if (! rq->expired_timestamp)
        rq->expired_timestamp = jiffies;
```

在这段代码中,首先对当前进程的 time_slice 参数进行判断,如果--time_slice 为0,即表示分配给此普通进程的时间片耗尽时,程序将进行以下操作:

1) 使用 dequeue_task 函数将此进程从 RQ 的 active 队列中摘除,该函数的详细描述见5.4.1节。

2) 使用 set_tsk_need_resched 函数将此进程状态改变为 TIF_NEED_RESCHED。

3) 使用 effective_prio 函数重新计算当前进程的 prio 参数,使用 task_timeslice 函数重新计算当前进程的 time_slice 参数,这两个函数的详细描述见5.1.2节。

4) 将 first_time_slice 参数置为0。

5) 如果 RQ 的 expired_timestamp 参数为0,则将 expired_timestamp 参数设置为当前时钟节拍 jiffies。expired_timestamp 参数为0表示在当前 RQ 中的 expired 队列中没有进程描述符。因为只有在 RQ 的 active 和 expired 对列交换时,expired_timestamp 参数才可能为0,此时 expired 队列中没有任何进程。

```
/* task_running_tick 函数源代码片段 4 */
      if (! TASK_INTERACTIVE(p) || expired_starving(rq)) {
        enqueue_task(p, rq->expired);
```

```
        if (p->static_prio < rq->best_expired_prio)
            rq->best_expired_prio = p->static_prio;
    } else
        enqueue_task(p, rq->active);
} else {
```

如果当前进程是交互式进程而且在 RQ 的 expired 队列中没有饿死的进程，则将当前进程加入到 RQ 中相应的 active 队列中。Linux 系统的调度程序优先在 active 队列中选择合适的进程，将交互式进程加入到 active 队列中有利于提高其响应度。

这段程序中使用宏 TASK_INTERACTIVE 判断当前进程是否为交互式进程。该宏的定义为：

```
#define TASK_INTERACTIVE(p) ((p)->prio <= (p)->static_prio - DELTA(p))
```

这里我们重温 5.1.2 节中 DELTA 的计算公式：

```
DELTA(p) = p->static_prio/4 -28
bonus = CURRENT_BONUS(p) - MAX_BONUS / 2
prio = p->static_prio - bonus
bonus = p->sleep_avg/100 - 5
```

由此可得布尔表达式 TASK_INTERACTIVE 的计算公式：

```
TASK_INTERACTIVE(p) <=> ((p)->prio <= (p)->static_prio - DELTA(p)) ⇔
((p)->prio <= (3 * p->static_prio/4 +28))                         ⇔
p->static_prio - bonus <= 3 * p->static_prio/4 +28                 ⇔
p->static_prio/4 < 28 + bonus                                      ⇔
p->static_prio/4 <= 23 + p->sleep_avg/100                          ⇔
p->static_prio <= 92 + p->sleep_avg/25
```

通过以上公式，可以发现宏 TASK_INTERACTIVE 只与进程的 static_prio 参数和 sleep_avg 参数有关，当一个进程的 sleep_avg 越大，其成为交互式进程的可能性越大。

这段程序使用 expired_starving 函数判断 RQ 的 expired 队列中是否有饿死的进程，该函数的源代码如下所示，如果该函数返回 1 表示 RQ 的 expired 队列中有饿死的进程。

```
static inline int expired_starving(struct rq *rq)
{
  if (rq->curr->static_prio > rq->best_expired_prio)
    return 1;
  if (! STARVATION_LIMIT || ! rq->expired_timestamp)
    return 0;
  if (jiffies - rq->expired_timestamp > STARVATION_LIMIT * rq->nr_running)
    return 1;
  return 0;
}
```

以上程序使用以下方法判断在 RQ 的 expired 队列中是否有饿死的进程。

● 如果在 expired 队列中，优先级别最高的进程大于当前进程的优先级别，表示在 expired

队列中有被饿死的进程。

- 如果 RQ 的 expired _ timestamp 参数为 0 时，表示 RQ 的 active 队列刚刚与 expired 队列进行交换，此时在 expired 队列中一定没有被饿死的进程。
- 如果 RQ 的 expired 队列重新建立的时间(jiffies - rq->expired _ timestamp)大于 RQ 中的进程数与宏 STARVATION _ LIMIT 之积时，表示 RQ 的 expired 队列中有被饿死的进程。在 Linux 系统中，宏 STARVATION _ LIMIT 等效与 MAX _ SLEEP _ AVG。

如果当前进程不是交互式进程或者在 RQ 的 expired 进程队列中有饿死的进程时，则将当前进程加入到 expired 进程队列中。随后这段程序比较 RQ 中的 best _ expired _ prio 参数将与当前进程的 static _ prio 参数，并使用较小的值更新 RQ 中的 best _ expired _ prio 参数。

Linux 系统采用这种方式管理 RQ 的 active 队列和 expired 队列保证交互式进程不能长时间在 active 队列中有效而饿死其他在 expired 队列中的进程。以上这几段源代码分析了普通进程使用完毕当前进程时间片后，task _ running _ tick 函数的处理过程。而当进程没有使用完毕当前时间片，task _ running _ tick 函数将执行以下源代码：

```
/* task_running_tick 函数源代码片段 5 */
    } else {
      if (TASK_INTERACTIVE(p) && !((task_timeslice(p) -
        p->time_slice) % TIMESLICE_GRANULARITY(p)) &&
        (p->time_slice >= TIMESLICE_GRANULARITY(p)) &&
        (p->array == rq->active)) {

        requeue_task(p, rq->active);
        set_tsk_need_resched(p);
      }
    }
out_unlock:
  spin_unlock(&rq->lock);
} /* End task_running_tick */
```

这段代码首先判断当前进程是否为交互式进程，如果不是，则跳转到 out _ unlock 标签处使用 spin _ unlock 打开自旋锁后然后返回。

如果当前进程是交互式进程，则进一步通过一些复杂的计算判断当前进程的 time _ slice 是否过大，如果该值过大则将此进程从 rq 的 active 相应队列中摘除。然后使用 set _ tsk _ need _ resched 函数将此进程状态改变为 TIF _ NEED _ RESCHED。Linux 系统禁止一个进程使用 time _ slice 参数过大，如果出现这种情况，Linux 系统将这个进程使用的 time _ slice 参数分解成一段一段符合大小的 time _ slice。

由以上代码，发现在每次调用 task _ running _ tick 函数时，都会将进程的 time _ slice 参数减 1，然后根据当前进程的时间片的使用情况对当前进程进行处理。在 task _ running _ tick 函数返回后，scheduler _ tick 函数和 timer _ interrupt 函数将会很快结束，同时准备退出系统时钟异常中断处理程序。

Linux 系统在退出系统时钟异常中断处理程序之前，将调用 ret _ from _ except 函数进行中断的返回，Linux PowerPC 大多数异常处理和中断处理结束后都会调用 ret _ from _ except

函数，在该函数中将有机会调用 schedule 函数实现进程的切换，ret _ from _ except 函数的详细介绍见第 6 章。

如果 scheduler _ tick 函数将一个进程的 flags 参数设置为 TIF _ NEED _ RESCHED 之后，该进程在当前系统时钟异常中断异常的 ret _ from _ except 函数中调用 schedule 函数被切换出去，但是在 scheduler _ tick 函数中，不会切换当前进程。

5.4.3 schedule 函数

schedule 函数是 Linux 进程调度的核心。在 Linux 系统中，schedule 函数的主要作用是在 RQ 的 active 队列中选择一个最合适的进程，然后对进程进行上下文切换，最后优先级最高的进程将被激活运行。

schedule 函数有两种调用方式，主动调用和被动调用。其中，主动调用是指在 Linux 系统的内核程序中直接调用 schedule 函数。这类情况通常发生在当前进程因为等待某些核心事件，如中断或者信号量时，将进程挂起，此时核心程序需要首先将当前进程的状态设置为 TASK _ INTERRUPTIBLE 或者 TASK _ UNINTERRUPTIBLE，然后再调用 schedule 函数切换当前进程。

而被动调用是指当前进程使用完当前时间片，或者 Linux 系统中有更高优先权的进程时，将 TIF _ NEED _ RESCHED 标志设置为 1，然后在系统调用、中断处理或者异常处理结束之后，由相应的回调函数隐式调用 schedule 函数切换当前进程。

在 Linux 系统中使用以下方式启动 schedule 函数。

(1) 在设备驱动程序、协议栈等其他内核程序的执行过程中，因为需要等待某些资源，主动调用 schedule 函数将当前进程切换。如上文提到的 wait _ event 函数可以调用 schedule 函数将进程主动进行切换。

(2) 在进程结束时，由 do _ exit 函数主动调用 schedule 函数，切换已经结束的进程。

(3) 系统调用结束时通过 ret _ from _ syscall ->syscall _ exit _ work ->ret _ from _ except ->do _ work 函数调用 schedule 函数。

(4) 在中断或者异常处理程序结束时通过 ret _ from _ except ->do _ work 函数调用 schedule 函数。

(5) 系统调用 nanosleep 也可以调用 schedule 函数切换进程。该系统调用可以迫使当前进程进入睡眠状态，然后在指定的时间里由操作系统将该进程唤醒。程序员常用的 sleep 库函数就是使用系统调用 nanosleep 实现的。

(6) 系统调用 pause 也可以调用 schedule 函数切换进程。该系统调用可以迫使当前进程进入睡眠状态，然后在接受某个信号后由操作系统将该进程唤醒。程序员常用的 pause 库函数就是使用系统调用 pause 实现的。该函数在 Linux 系统中使用 sys _ pause 函数实现。

(7) 有时一个进程由于某种原因需要放弃 CPU 资源时，还可以调用 yield 函数主动让出 CPU 资源，yield 函数也会调用 schedule 函数。yield 函数不同于 wait _ event 函数，该函数仅让出当前进程使用的 CPU 资源，而不将当前进程放入进程的等待队列中。该函数的源代码如下：

```
void _ sched yield(void)
{
  set _ current _ state(TASK _ RUNNING);
  sys _ sched _ yield();
}
```

该函数首先将进程状态设置为 TASK_RUNNING，然后调用 sys_sched_yield 函数迫使当前进程让出 CPU 资源。sys_sched_yield 函数的源代码如下：

```
asmlinkage long sys_sched_yield(void)
{
  struct rq *rq = this_rq_lock();
  struct prio_array *array = current->array, *target = rq->expired;
...
  if (rt_task(current))
    target = rq->active;
...
  if (array != target) {
    dequeue_task(current, array);
    enqueue_task(current, target);
  } else
    requeue_task(current, array);
  _release(rq->lock);
  spin_release(&rq->lock.dep_map, 1, _THIS_IP_);
  _raw_spin_unlock(&rq->lock);
  preempt_enable_no_resched();

  schedule();
  return 0;
} /* End sys_sched_yield */
```

通过以上程序可以发现，当前进程为实时进程时，该进程将被加入到 RQ ->active 队列的尾部，否则将被加入到 RQ ->expired 队列的尾部，之后该函数调用 schedule 函数将当前进程切换出去。

Linux PowerPC 调用 schedule 函数后，将进行一系列操作完成进程的切换。Linux 系统的 schedule 函数编写的逻辑性较强。如果读者真正理解了本章的以上内容，阅读这段代码的难度应该不大。schedule 函数分为两个部分：进程切换前的准备和进程的切换。

1. 进程切换前的准备

schedule 函数没有输入参数也没有返回值，只使用系统的全局变量进行输入输出操作。

```
asmlinkage void _sched schedule(void)
{
  struct task_struct *prev, *next;
  struct prio_array *array;
  struct list_head *queue;
  unsigned long long now;
  unsigned long run_time;
  int cpu, idx, new_prio;
  long *switch_count;
  struct rq *rq;
```

schedule 函数在开始处定义了一些临时变量，其中 prev 和 next 含义如下：

- prev 变量用来保存 current 指针，指向当前进程描述符。
- next 变量将指向被选中的进程，进程调度结束后该进程将取代当前进程继续执行。如果系统中没有优先权高于当前进程的进程，next 变量将与 current 变量相等，此时不会切换当前进程。

```
if (unlikely(in_atomic() && ! current->exit_state)) {
    printk(KERN_ERR "BUG: scheduling while atomic: "
        "%s/0x%08x/%d\n",
        current->comm, preempt_count(), current->pid);
    debug_show_held_locks(current);
    if (irqs_disabled())
        print_irqtrace_events(current);
    dump_stack();
}
profile_hit(SCHED_PROFILING, __builtin_return_address(0));
```

schedule 函数在进行一系列参数检查后将调用 profile_hit 函数。profile_hit 函数是一个与内核性能测试分析有关的函数。Linux 系统通过 make menuconfig 配置 Linux 源代码时，将 Instrumentation Support→profile 选项使能以支持 profile 工具。在内核支持 profile 工具后，可以在 Linux 内核启动时加入 profile = 1 参数使用这一工具。

Linux 内核在启动后将会创建/proc/profile 文件，通过读取这些文件，可以获得有关处理器运行的许多信息。根据 Linux 系统的启动配置不同，读取/proc/profile 文件的结果有所不同。如果 Linux 启动参数被设置为"profile = schedule? "，读取 profile 文件可以获得每个函数调用 schedule 函数的次数，这一点用来调试 schedule 十分有用。

对此功能有兴趣的读者可以详细阅读 ./kernel/profile.c 文件和/driver/oprofile 目录下的相关文件。在 schedule 函数中，profile_hit 函数的作用是将当前指令的计数器加 1。Linux 系统可以根据此计数器计算出调用 schedule 函数的次数。

所有 profiling 类工具都有一个共同的特点，就是所分析的数据总是或多或少有些误差，但是这些有误差的数据仍然可以在很大程度上反映系统的运行状态。中级程序员向高级程序员进阶的必经之路就是能够合理使用 profiling 工具对程序的性能准确地进行分析，对程序的整体性能进行评估，从而找到瓶颈所在。

然而实现这一点其实并不容易。同一片树叶，有的人一叶障目，有的人一叶知秋。

掌握 Profiling 工具并不容易，有关 Profiling 工具的讨论远远超出了本书的范围，对此有兴趣的读者可以阅读一下 Glenn Ammons，Thomas Ball，James R. Larus 的 Exploiting hardware performance counters with flow and context sensitive profiling 和 J. M. Anderson, W. E. Weihl, L. M. Berc, J. Dean, S. Ghemawat 的 Continuous profiling: where have all the cycles gone? 等文章去学习有关 Profiling 的技巧。

```
need_resched:
    preempt_disable();
    prev = current;
```

```
    release_kernel_lock(prev);
need_resched_nonpreemptible:
    rq = this_rq();
...
    schedstat_inc(rq, sched_cnt);
    now = sched_clock();
```

schedule 函数首先将禁止内核抢占，然后将 current 变量保存在 prev 变量中以便进行进程切换，之后获得 rq 和当前时间 now 的值。

```
    if (likely((long long)(now - prev->timestamp) < NS_MAX_SLEEP_AVG)) {
        run_time = now - prev->timestamp;
        if (unlikely((long long)(now - prev->timestamp) < 0))
            run_time = 0;
    } else
        run_time = NS_MAX_SLEEP_AVG;
```

该段程序将当前 CPU 时间减去 prev 进程的最近一次使用 CPU 的时间戳 timestamp 参数，从而得到当前进程使用 CPU 的时间 run_time。Linux 系统中 run_time 的值不能大于 NS_MAX_SLEEP_AVG。

进程的 timestamp 参数可以在以下情况下被更新：

（1）当进程刚被创建或者从等待状态加入到 RQ 的 active 相应队列时，Linux 系统将使用 activate_task 函数更新 timestamp 参数。此时通过 now - p ->timestamp 可以得到当前进程的睡眠时间。

（2）当正在运行的进程 p1 被 schedule 函数切换到进程 p2 时，p2 进程的 timestamp 参数将被更改为当前时间 now，p2 进程也将成为新的当前进程。此后当 p2 进程第一次进入 schedule 函数时，通过 now - p2 ->timestamp 就会得到当前进程 p2 使用 CPU 的时间。在上面这段程序中，进程的 run_time 参数就是基于此计算出来的。

在 Linux 系统中，进程的 timestamp 参数十分重要，该参数在进程创建和切换时都将被更改，timestamp 参数对于 Linux 系统进程调度的时间参数计算十分重要。在 Linux 系统进程调度中，一个重要细节就是如何对时间的精确计算。进程调度中一些大的算法表面上看是比较容易理解的，只是准确理解这些算法需要理解许多细节知识。

```
    /*
     * Tasks charged proportionately less run_time at high sleep_avg to
     * delay them losing their interactive status
     */
    run_time /= (CURRENT_BONUS(prev) ? : 1);

    spin_lock_irq(&rq->lock);
```

以上程序根据进程的 bonus 的值按比例缩小 run_time 的值，随后使用 spin_lock_irq 获得 rq 的自旋锁。spin_lock_irq 与 spin_lock 函数不同，该函数在获得自旋锁的同时屏蔽外部中断。

```
switch_count = &prev->nivcsw;
if (prev->state && ! (preempt_count() & PREEMPT_ACTIVE)) {
  switch_count = &prev->nvcsw;
  if (unlikely((prev->state & TASK_INTERRUPTIBLE) &&
      unlikely(signal_pending(prev))))
    prev->state = TASK_RUNNING;
  else {
    if (prev->state == TASK_UNINTERRUPTIBLE)
      rq->nr_uninterruptible++;
    deactivate_task(prev, rq);
  }
}
```

以上程序对 prev 进程进行状态检查,如果 prev 进程的 state 参数不是 TASK_RUNNING 而且 prev 进程不是被因为被抢占而进入 schedule 函数的,则继续判断 prev 进程的状态是否为 TASK_INTERRUPTIBLE。

如果 prev 进程是因为等待某些信号量而使用 schedule 函数主动进行切换,同时在执行到本段代码时,等待的信号正巧来临,即 signal_pending(prev)的结果为真,则将 prev ->state 的值置为 TASK_RUNNING,发生这种情况的概率非常小;否则继续判断 prev ->state 参数的值是否为 TASK_UNINTERRUPTIBLE,如果是则将 RQ 的 nr_uninterruptible 计数加 1,然后将 prev 进程从进程的活动队列中摘除。

当处理进程主动地将进程描述符中的 state 改写为 TASK_UNINTERRUPTIBLE 或者 TASK_INTERRUPTIBLE,然后使用 schedule 函数将该进程切换时,将运行以上代码。

```
cpu = smp_processor_id();
if (unlikely(! rq->nr_running)) {
  idle_balance(cpu, rq);
  if (! rq->nr_running) {
    next = rq->idle;
    rq->expired_timestamp = 0;
    wake_sleeping_dependent(cpu);
    goto switch_tasks;
  }
}
```

这段程序首先获得当前进程在使用的 CPU。如果 RQ 中没有处于 TASK_RUNNING 状态的进程则调用 idle_balance 函数和 wake_sleeping_dependent 函数。这两个函数只对 Linux SMP 系统有效,对于单 CPU 的 Linux 系统中这两个函数为空函数。如果 idle_balance 函数没有将在其他 CPU 中的进程搬移到当前 RQ 时,rq ->nr_running 将仍然为 0,此时将 next 指向 idle 进程,并将 RQ 的 expired_timestamp 参数置为 0,wake_sleeping_dependent 函数用来支持 SMT 体系的处理器,本书对此不做介绍。在单处理器系统中以上代码较为简单,当 RQ 中没有处于 TASK_RUNNING 状态的进程,则选择 idle 进程作为下一个即将运行的进程,然后进行进程切换。

```
array = rq->active;
if (unlikely(! array->nr_active)) {
  /*
   * Switch the active and expired arrays.
   */
  schedstat_inc(rq, sched_switch);
  rq->active = rq->expired;
  rq->expired = array;
  array = rq->active;
  rq->expired_timestamp = 0;
  rq->best_expired_prio = MAX_PRIO;
}
```

如果 RQ 的 active 队列中没有活动进程,则将指向 RQ 中的 active 队列的指针与指向 RQ 中 expired 队列的指针互换。这两个指针互换完毕后,RQ 的 expired 队列将不会存在任何进程,因此这段程序将 RQ 的 expired_timestamp 参数和 best_expired_prio 参数分别置为 0 和 MAX_PRIO。

```
idx = sched_find_first_bit(array->bitmap);
queue = array->queue + idx;
next = list_entry(queue->next, struct task_struct, run_list);
```

这段程序是进程调度的关键代码。这段程序用来确定在当前系统中优先权最高的进程,其执行流程如下所示:

1) 在 rq ->active ->bitmap 中查找第一个有效位 idx。

2) 使用 idx 获得在当前 active 中优先权最高的队列 queue。

3) 获得这个 queue 中的第一个进程 next。Linux 系统认为这个进程就是当前 Linux 系统中优先权最高的进程。

在获得当前系统优先权最高的进程后,将执行以下程序:

```
if (! rt_task(next) && interactive_sleep(next->sleep_type)) {
  unsigned long long delta = now - next->timestamp;
  if (unlikely((long long)(now - next->timestamp) < 0))
    delta = 0;

  if (next->sleep_type == SLEEP_INTERACTIVE)
    delta = delta * (ON_RUNQUEUE_WEIGHT * 128 / 100) / 128;

  array = next->array;
  new_prio = recalc_task_prio(next, next->timestamp + delta);

  if (unlikely(next->prio != new_prio)) {
    dequeue_task(next, array);
    next->prio = new_prio;
```

```
            enqueue_task(next, array);
        }
    }
    next->sleep_type = SLEEP_NORMAL;
```

以上代码主要执行以下操作:

1) 如果 next 进程不是实时进程而且该进程的 sleep_type 参数是 SLEEP_INTERACTIVE 或者为 SLEEP_INTERRUPTED 时,将重新计算 next 进程的 prio 和 sleep_avg 参数。

2) delta 变量记录 next 进程在 RQ 队列中的等待时间。如果进程的 sleep_type 参数为 SLEEP_INTERACTIVE 时,delta 的值为 3 * delta/10。

3) 根据 delta 的值重新计算进程的优先级别 new_prio,如果 next 进程的 prio 与 new_prio 不相等则将 next ->prio 置为 new_prio。

4) 最后将进程的 sleep_type 参数更改为 SLEEP_NORMAL。

要深刻理解这段代码需要了解进程描述符的 sleep_type 参数。为明晰起见,Linux 系统使用 sleep_type 参数替换了版本较早的 Linux 的进程描述符的 activated 参数,对此有兴趣的读者可以参考以下 URL:

http://lkml.org/lkml/2006/1/13/75

由 5.1.2 节得知,进程描述符的 sleep_type 参数共有以下四种状态:

(1) SLEEP_NONINTERACTIVE,该状态与 activated 参数为-1 的值相对应。表示进程从 TASK_UNINTERRUPTIBLE 等待状态中激活。

(2) SLEEP_NORMAL,该状态与 activated 参数为 0 的值相对应。表示进程一直处于就绪态,没有因为等待某个中断或者信号资源而进入过等待状态。

(3) SLEEP_INTERACTIVE,该状态与 activated 参数为 1 的值相对应。表示进程从 TASK_INTERRUPTIBLE 等待状态中激活。而且激活该进程的程序不运行在中断处理程序中。

(4) SLEEP_INTERRUPTED,该状态与 activated 参数为 2 的值相对应。表示进程从 TASK_INTERRUPTIBLE 等待状态中激活。而且激活该进程的程序运行在中断处理程序中。

在 Linux 系统中,一个进程的 sleep_type 参数可以按照以下规则进行转换:

(1) 一个进程第一次加入到进程的 RQ 中时,其进程描述符的 sleep_type 参数与其父进程 sleep_type 参数相同。进程第一次获得 CPU 资源运行时,其进程描述符的 sleep_type 参数将被改写为 SLEEP_NORMAL。

(2) 当进程使用完当前的时间片,被切换到就绪状态时,其 sleep_type 参数不变。当这个进程再次从就绪到运行时,其进程描述符的 sleep_type 参数将被置为 SLEEP_NORMAL。

(3) 当进程因为等待某些信号资源,被切换到等待状态时,其 sleep_type 参数不变。但是当这个进程等待的信号来临时,Linux 系统将调用 try_to_wake_up 函数将其 sleep_type 参数改变为 SLEEP_NONINTERACTIVE。

(4) 当进程因为等待某些中断资源,被切换到等待状态时,其 sleep_type 参数不变。但是当这个进程等待的中断来临时,Linux 系统将调用 try_to_wake_up→activate_task 函数将进程从等待状态迁移到就绪状态,同时将 sleep_type 参数改变为 SLEEP_INTERACTIVE (在中断处理程序中调用 activate_task 函数)或者 SLEEP_INTERRUPTED(没有在中断处理

程序中调用 activate _ task 函数)。

由此可见,在 sleep _ type 参数的四种状态中,处于 SLEEP _ INTERRUPTED 状态的进程交互式等级最高,因为 Linux 系统认为经常需要被中断程序唤醒的进程,其交互式等级最高;处于 SLEEP _ INTERACTIVE 和 SLEEP _ NONINTERACTIVE 状态的进程,其交互式等级次之;处于 SLEEP _ NORMAL 状态的进程,其交互式等级最低。

Linux 系统并不直接使用以上四种状态判断该进程是否为交互式进程,而是通过这些状态影响进程的 sleep _ avg 参数,从而间接地影响当前进程的交互性。当一个交互式进程较长时间地使用 CPU 后,该进程的交互式级别将逐渐降低,从而转换成为一个普通进程,这也是在本段程序中将 delta 的值改变为 3 * delta/10 的原因。

2. 进程的切换

Linux 系统选择出优先权最高的进程后,将进程 next 与当前进程 prev 的进程运行空间切换,其代码如下:

```
switch_tasks:
  if (next == rq->idle)
    schedstat_inc(rq, sched_goidle);
  prefetch(next);
  prefetch_stack(next);
  clear_tsk_need_resched(prev);
  rcu_qsctr_inc(task_cpu(prev));
  update_cpu_clock(prev, rq, now);
```

这段代码首先调用 prefetch 函数,将 next 进程的进程描述符指针预取到 Cache 中。Linux PowerPC 使用 dcbt 指令实现这一操作,目前 Linux 系统对 Cache 的优化只针对 L1 Cache。之后这段代码将进程描述符 flags 参数的 TIF _ NEED _ RESCHED 位清除。

```
prev->sleep_avg -= run_time;
if ((long)prev->sleep_avg <= 0)
  prev->sleep_avg = 0;
prev->timestamp = prev->last_ran = now;

sched_info_switch(prev, next);
```

这段程序根据进程使用的时间片 run _ time 减少 prev 进程的平均睡眠时间 sleep _ avg,然后更新 prev 进程的 timestamp 和 last _ ran 参数为 now。

```
...
  if (likely(prev != next)) {
    next->timestamp = now;
    rq->nr_switches++;
    rq->curr = next;
    ++*switch_count;

    prepare_task_switch(rq, next);
    prev = context_switch(rq, prev, next);
    barrier();
```

```
        /*
         * this_rq must be evaluated again because prev may have moved
         * CPUs since it called schedule(), thus the 'rq' on its stack
         * frame will be invalid.
         */
        finish_task_switch(this_rq(), prev);
    } else
        spin_unlock_irq(&rq->lock);

    prev = current;
...
} /* End Schedule */
```

这段程序首先判断 prev 参数是否与 next 参数相等。当 RQ 的 active 队列中所有进程的优先权都不高于 prev 进程时，prev 与 next 相等，此时不需要进行进程切换。但是在多数情况下，prev 与 next 的值不相同，此时必须进行进程切换，其步骤如下：

1）首先这段程序更新 next 进程的 timestmap 参数为 now。当 next 进程被 schedule 函数切换后，可以利用在这里保存的 timestamp 参数计算 next 进程在 CPU 中运行的时间。

2）随后更新 RQ 中的 nr_switches 和 curr 参数，确定 RQ 中的当前进程为 next 进程。

3）调用 context_switch 函数将 prev 与 next 进程进行切换，该函数是进程切换的要点。该函数将完成新老进程的运行空间的切换，该函数将在下文详细描述。

4）执行存储器栏栅操作保证进程切换后的内存一致性，然后调用 finish_task_switch 函数清理进程切换后的现场。

3. context_switch 函数

context_switch 函数是进程切换中最重要的函数，其主要作用有两个，一是切换 prev 进程和 next 进程使用的内存空间，二是切换 prev 进程和 next 进程的上下文，包括这两个进程使用的系统寄存器，堆栈等系统资源，其源代码如下：

```
static inline struct task_struct *
context_switch(struct rq *rq, struct task_struct *prev,
        struct task_struct *next)
{
```

在 Linux 系统中，context_switch 函数只能被 schedule 函数调用，该函数中的三个参数与 schedule 函数中的 rq，prev 和 next 变量的值相同，返回值为被切换进程的进程描述符。

```
struct mm_struct *mm = next->mm;
struct mm_struct *oldmm = prev->active_mm;

if (! mm) {
  next->active_mm = oldmm;
  atomic_inc(&oldmm->mm_count);
  enter_lazy_tlb(oldmm, next);
} else
  switch_mm(oldmm, mm, next);
```

要理解这段程序，读者需要对 Linuix 内存管理有一定的认识。读者可以在阅读本书的第7章后，重新温习这段代码。这段代码的执行流程如下：

1）这段程序将首先判断 next 进程的 mm 参数是否为 NULL。如果为空，表示 next 进程是核心进程。此时将 next 进程的 active_mm 参数置为 prev→active_mm 参数，表示核心进程 next 将借用 prev 进程的地址空间。

2）prev->active_mm->mm_count 参数加1，以增加对此用户进程地址空间的引用计数。

3）调用 enter_lazy_tlb 函数。对于基于 E500 内核的 Linux PowerPC，该函数为空函数，不过有的体系结构的处理器支持这个函数。Lazy TLB 的主要作用是提高 SMP 结构的处理器系统或者 DSM 结构的处理器系统的 TLB 的刷新效率，对此有兴趣的读者可以阅读 Ernest S. Cohen 的 Lazy Flushing of Translation Lookaside Buffers，这篇文章提供了一种使用 Lazy TLB 的方法。这个方法也是一个专利技术(US Patent 7,069,389)。

4）如果 next 进程的 mm 参数不为 NULL，则表示 next 进程是一个用户进程，此时将调用 switch_mm 函数完成进程地址空间的转换。

switch_mm 函数在 ./include/asm-powerpc/mmu_context.c 文件中。在 switch_mm 函数中，如果 prev 与 next 相等，将不做任何操作，否则将调用 switch_stab 函数将 next 进程的 mm 描述符、PC(Program Counter)和堆栈指针 SP 预取到 sdata 段的 stab 中，在 stab 中存放着数据的物理地址与虚拟地址的转换关系，其结构如下所示：

```
struct stab_entry {
    unsigned long esid_data;
    unsigned long vsid_data;
};
```

switch_stab 函数调用_ste_allocate ->make_ste 函数在 sdata 段中建立当前 PC 和 SP 所在物理地址的 stb_entry。Linux PowerPC 使用寄存器 GPR13 保存 sdata 段，sdata 段内数据或者指令的访问与 data 段和 text 段的访问机制有所不同，该函数还会完成新老进程的 TLB 同步。switch_mm 函数执行完毕后，context_switch 函数将执行以下代码。

```
if (! prev->mm) {
    prev->active_mm = NULL;
    WARN_ON(rq->prev_mm);
    rq->prev_mm = oldmm;
}
```

这段程序对 prev ->mm 参数进行检查。如果其值为 NULL，则表示 prev 进程是核心进程，此时需要将 prev 进程借用的进程地址空间 active_mm 释放，随后将 RQ 的 prev_mm 参数置为 oldmm。在 context_switch 函数的最后，将调用 switch_to 函数，切换新老进程使用的系统寄存器和堆栈：

```
    switch_to(prev, next, prev);
} /* End context_switch */
```

Linux PowerPC 的 switch 函数在 ./include/asm-powerpc/system.h 文件中，Linux 系统使用宏定义实现该函数：

```
#define switch_to(prev, next, last) ((last) = _switch_to((prev), (next)))
```

宏 switch_to 将调用_switch_to 函数，_switch_to 函数在 ./arch/powerpc/kernel/process.c 文件中，其源代码如下：

```
struct task_struct *_switch_to(struct task_struct *prev, struct task_struct *new)
{
    struct thread_struct *new_thread, *old_thread;
    unsigned long flags;
    struct task_struct *last;

    new_thread = &new->thread;
    old_thread = &current->thread;

    local_irq_save(flags);

    last = _switch(old_thread, new_thread);

    local_irq_restore(flags);
    return last;
}
```

这段程序使用_switch 函数切换新老进程使用的系统寄存器和堆栈。

4. _switch 函数

_switch 函数在 ./arch/powerpc/kernel/entry_32.S 文件中，该函数将 next ->thread 参数和 prev ->thread 参数交换。该函数有两个输入参数 old_thread 和 new_thread。在_switch_to 函数中，可以发现 old_thread = &current ->thread，即当前进程 thread 参数。而 new_thread = &new ->thread，即为新进程 thread 参数的地址。

在_switch 函数执行完毕后，current 指针将被更新，新老进程的上下文在这个函数返回时被切换。_switch 函数返回后，将不在 prev 进程的上下文中执行而是在 next 进程的上下文中执行，这个函数是 Linux PowerPC 实现进程切换的核心。

```
_GLOBAL(_switch)
    stwu r1,-INT_FRAME_SIZE(r1)
    mflr r0
    stw r0,INT_FRAME_SIZE+4(r1)
    SAVE_NVGPRS(r1)
    stw r0,_NIP(r1)      /* Return to switch caller */
    mfmsr r11
    li r0,MSR_FP         /* Disable floating-point */
    and. r0,r0,r11       /* FP or altivec or SPE enabled? */
    beq+ 1f
    andc r11,r11,r0
    MTMSRD(r11)
    isync
```

这段代码的执行流程如下：

1) 建立_ switch 函数的栈帧,该函数堆栈大小为 INT _ FRAME _ SIZE。

2) 将 LR 寄存器存入寄存器 GPR0 中,然后将寄存器 GPR0 压入堆栈,以便使用 blr 指令将_ switch 函数返回。

3) 使用 SAVE _ NVGPRS 将寄存器 GPR13~31 存入当前栈帧的相应位置。_ switch 函数中没有临时变量,因此当前堆栈从 GPR1 + 16(STACK _ FRAME _ OVERHEAD)开始存放 pt _ reg 结构中 32 位寄存器。由 2.4.2 节得知,寄存器 GPR3~10 用来存放系统的输入参数,所以不用保存。

4) 将寄存器 GPR0 的值存入到堆栈的相应位置。

5) 将 MSR 寄存器存入 GPR11 寄存器中。

6) 识别当前处理器是否使能了 FP 处理器、Altivec 或 SPE。Freescale 的 G4 系列的处理器内含有 Altivec,而 E500 内核的处理器含有 SPE。

```
1: stw r11,_MSR(r1)
   mfcr r10
   stw r10,_CCR(r1)
   stw r1,KSP(r3)      /* Set old stack pointer */
```

- 将 GPR11 寄存器压入堆栈。
- 将 CR 寄存器保存到 GPR10 寄存器中,然后再压入堆栈。以上寄存器的压栈方法与 PowerPC ABI 规定的栈帧结构完全兼容,读者可以回顾 2.4.3 节进一步了解 PowerPC 处理器的堆栈结构。

```
tophys(r0,r4)
CLR_TOP32(r0)
mtspr SPRN_SPRG3,r0      /* Update current THREAD phys addr */
lwz r1,KSP(r4)           /* Load new stack pointer */
```

- 将保存在 GPR4 寄存器中的 new _ thread 的虚拟地址通过宏 tophys 转换为物理地址,然后将这个物理地址存放到 GPR0 寄存器中。E500 内核的 MMU 不能关闭,因此存放在 GPR0 寄存器的地址还是虚拟地址。
- 将 GPR0 寄存器的值存放到寄存器 SPRG3 中。Linux PowerPC 中使用寄存器 SPRG3 存放当前进程描述符的 thread 参数。
- 使用 new _ thread 中保存的堆栈指针寄存器更新寄存器 GPR1,完成新老进程堆栈空间的切换。

```
mr r3,r2
addi r2,r4,-THREAD      /* Update current */
```

- 将当前进程描述符的地址存入寄存器 GPR3 中。在_ switch 函数返回时,GPR3 寄存器中保存返回值。由此可以发现,_ switch 函数的返回值为老进程的进程描述符,即 prev 的进程描述符指针。
- 将寄存器 GPR4 的内容减去 THREAD,其中 GPR4 寄存器中存放着即将获得 CPU 资源运行进程 thread 参数的地址,而宏 THREAD 用来记录 thread 结构在 task _ struct 结构中的偏移。因此两者之差为即将获得 CPU 资源运行进程的进程描述符的地址。至此,

该函数完成了新老进程的进程描述符的切换，新进程的描述符地址将被存入 GPR2 寄存器中。此时 Linux 系统已经认为，当前进程为 next 进程。

```
lwz r0,_CCR(r1)
mtcrf 0xFF,r0
/* r3-r12 are destroyed -- Cort */
REST_NVGPRS(r1)

lwz r4,_NIP(r1) /* Return to _switch caller in new task */
mtlr r4
```

从堆栈中恢复 CR，GPR13～31 和 LR 寄存器的内容。此时读者需要明白此时恢复的 CR，GPR13～31，LR 寄存器不是 prev 进程进入_switch 函数时存放的值，而是 next 进程进入_switch 函数时存放的值。因为在此之前，_switch 函数已经切换了新老进程的堆栈。

next 进程是本次进程调度的受益者，这个进程即将获得 CPU 运行。但是该进程也曾经被其他进程切换过，那时该进程将在进入_switch 函数时将 CR，GPR13～31 和 LR 寄存器保存起来，此时恢复的 CR，GPR13～31 和 LR 寄存器就是当时保存在堆栈里的寄存器。

```
addi r1,r1,INT_FRAME_SIZE
blr
```

首先调整堆栈指针 GPR1，然后使用 blr 指令将进程跳转到 LR 寄存器指定的地址处运行，并将保存在 GPR3 寄存器中的进程描述符作为返回值。当_switch 函数返回时，已物似人非，_switch 函数由 prev 进程调用，但是返回时将运行在 next 进程的地址空间中。

上述程序段的 blr 指令所在的程序地址是一个非常特殊的程序地址，当一个旧进程因为某种原因被切换时，旧进程就停留在这个程序地址中，下一次运行时也是由此程序地址开始重新执行。在 Linux 系统中，所有处于等待、就绪状态的进程都停留在这条 blr 指令所在的程序地址处，这条指令所在的地址也是进程切换的转折点，在此之前该程序使用 prev 进程的上下文，而在此之后将使用 next 进程的上下文。

_switch 函数返回后，_switch_to 函数也很快执行完毕，从而完成 context_switch 函数的执行、进程的切换。

第 6 章　Linux PowerPC 的外部中断处理系统

Linux PowerPC 的中断系统与进程调度、内存管理及设备驱动程序紧密相连。本书在第 5 章中，介绍了系统时钟异常的处理，下文将介绍 Linux PowerPC 对外部中断的处理和系统调用异常。在 PowerPC 处理器中，当外部中断事件发生时，处理器停止执行当前指令，并跳转到外部中断处理程序中执行。Linux PowerPC 在设计外部中断处理程序时，需要考虑以下内容，以保证外部中断系统设计的完备性。

- 在 Linux PowerPC 中，设备驱动程序的中断处理服务例程最终挂接到外部中断处理函数 ExternalInput 中，统一进行处理，如何合理地设置数据结构处理这些来自不同设备驱动程序的中断请求？
- Linux PowerPC 需要保证中断处理程序能够正常运行，能够使不同种类的设备驱动程序共享同一个中断信号 int#，如何实现这种共享？
- 操作系统支持中断嵌套，如何实现这一机制？
- 在 Linux PowerPC 的中断处理程序中，并不是所有代码都支持中断重入，如何界定哪些代码可以被中断重入？
- 在外部中断处理程序中，需要调用其他函数，也需要支持中断嵌套。因此外部中断处理程序需要使用堆栈，保存一些必要的参数和中间结果。如何在外部中断处理程序中设置中断堆栈也是系统程序员需要认真考虑的问题。

在外部中断处理的过程中，除了注意以上问题之外，程序员还需要注意 Linux PowerPC 外部中断处理程序的优先权较高。程序员在书写外部中断处理程序的代码时，需要尽可能精炼，尽可能减少中断处理程序占用的 CPU 时间。

Linux PowerPC 外部中断处理系统由三大部分组成：

(1) Linux PowerPC 对外部中断系统的初始化，包括为一些重要数据结构空间的分配内存并进行初始化。Linux PowerPC 在进入 ExternalInput 函数处理外部中断之前，需要进行必要的准备工作。

(2) 设备驱动程序的中断服务例程与外部中断处理系统的挂接。在设备驱动程序初始化时，使用 request_irq 函数将外部设备的中断服务例程与 Linux PowerPC 的 ExternalInput 函数挂接。进入 ExternalInput 函数后，Linux PowerPC 经过一系列的操作后，将调用相应设备驱动程序的中断例程。

(3) 外部中断的处理，即外部中断处理函数 ExternalInput 的实现过程。在 ExternalInput 函数中，除了要执行中断服务例程外，还需要为 Linux PowerPC 进入中断进行必要的准备工作，如保存一些系统寄存器，设立中断堆栈等。ExternalInput 函数返回时，Linux PowerPC 根据中断执行的结果，决定是否需要进行进程调度。

Linux 的外部中断处理系统与进程调度紧密相连。在介绍 Linux PowerPC 外部中断处理系统的三大组成部分之前，我们需要首先介绍 PowerPC E500 内核的外部中断。

下文以 MPC8541 处理器为例，说明外部中断处理程序的流程和一些与外部中断有关的

基本概念，之后介绍 Linux PowerPC 如何使用 MPC8541 处理器提供的这些功能实现外部中断处理系统。

本章内容较为复杂，对 PowerPC 处理器中断系统和 Linux 系统没有一定基础的读者，会觉得难以理解。希望读者坚持阅读一遍本章的全部内容，再经过多次反复，彻底理解本章的全部内容，而不要半途而废。

6.1 MPC8541 处理器的中断系统

MPC8541 处理器的中断系统由两部分组成，一是 E500 内核的中断及异常的处理；二是 OpenPIC 中断控制器。

在 E500 内核中，包含两种可以暂时中止处理器运行当前指令的事件，中断和异常。其中，异常是由 E500 内核产生的，如出现非法指令，访问存储器时出现 TLB Miss 等情况；而中断通过处理器内核的外部引脚，如 int#，cint# 和 mcp 信号有效时，产生的事件。

从广义上说，在 E500 内核中，中断也是异常的一种。而从狭义上说，PowerPC 体系的中断特指处理器内核的 int#，cint# 和 mcp 信号有效时，产生的异常。为明晰起见，本书对中断和异常重新进行定义。

在本书中，由处理器内核产生的中断事件定义为异常，而由 int#，cint# 和 mcp 信号有效时产生的异常称为外部中断。在 E500 内核中，中断和异常都可以引发处理器进行状态切换。当这类事件发生后，PowerPC 系统将停止目前正在执行的代码，而转去执行中断或者异常服务程序。

E500 内核中，外部中断分为 Noncritical 中断、Critical 中断和 Machine Check 中断，这些中断由 E500 内核的输入信号触发。

int#、cint#、core_fault_in# 信号为低时，E500 内核触发 Noncritical 中断或者 Critical 中断；mcp 信号出现上升沿时，E500 内核触发 Machine Check 中断。在 E500 内核中，中断的处理顺序分别为 mcp# 类中断、cint# 类中断、int# 类中断。

- int# 信号有效时，表示有外部中断事件发生。在 PQIII 系列的处理器中，这些外部中断由中断控制器 PIC(Programmable Interrupt Controller)管理。这些外部中断包括外部中断引脚 IRQ 引发的中断、PCI 总线控制器引发的中断、TSEC 中断控制器以及所有 E500 内核之外的设备引发的中断。当 MSR 寄存器的 EE 位为 1 且 int# 信号有效时，E500 内核将处理这些中断。
- cint# 信号有效时，表示有 critical 中断事件发生。cint# 的中断源与 int# 一致，都是来自中断控制器 PIC。在 PIC 中有一组寄存器 IIDRn，当 IIDRn 的 CI 位为 1 时，表示相应的中断信息使用 int# 信号发向 E500 内核，为 1 时表示相应的中断信息使用 cint# 信号发向 E500 内核。当 MSR 寄存器的 CE 位为 1 且 cint# 信号有效时，E500 内核将处理这些中断。
- mcp# 出现上升沿时，表示有 Machine Check 中断事件发生。当 MSR 寄存器的 ME 位为 1 并且 mcp# 信号有效时，MPC8541 将处理此中断。
- core_fault_in# 信号有效时，表示系统总线进行读写操作时发生错误。此时软件可以通过设置，由 Machine Check 中断处理程序处理这类错误。

E500 内核的异常是由 E500 内核内部的事件引发,如下所示。

- E500 内核执行非法指令时会产生异常。一般来说,编译器所编译出来的指令都是合法指令。但是,有时因为存放程序的 DDR 或者 FLASH 的硬件设计或者配置不正确,而导致在处理器执行程序时,所获得的指令为非法指令。
- E500 内核在用户模式下,访问只有在超级模式下才能访问的寄存器。
- 访问 E500 内核中未定义的特殊寄存器。
- 访问未定义的虚存空间,或者非对界的访问。
- E500 内核执行 sc,tw 或者 twi 指令时,引发的系统调用和 trap 中断。

E500 内核规定了异常的优先权,优先权高的异常可以中断低优先权的异常处理程序;而低优先权的异常需要等待高优先权的异常处理程序执行完毕。E500 内核中异常的优先权如下所示:

- HRESET。不同于 603E 内核,E500 内核没有将 HRESET#信号有效时引发的各种操作定义为异常,在 E500 内核中,没有为复位异常准备专门的中断向量。
- Machine Check 异常。机器自检异常。
- Critical Interrupt 异常。该异常由中断控制器 PIC 产生,用于外部中断,使用 cint#信号将中断信息传递给 E500 内核。
- Debug Interrupt 异常。该异常使用 E500 内核的 Critical Interrupt 的中断向量,用于系统调试。
- External Input 中断。该异常由中断控制器 PIC 产生,用于外部中断,使用 int#信号将中断信息传递给 E500 内核。在本书中,该异常被称作外部中断。
- ITLB Miss 异常。当处理器进行取指操作时,所访问的指令地址没有在 TLB0 和 TLB1 中命中时 E500 内核产生此异常。
- ISI 异常。在用户模式时,访问在超级模式下才能访问的内存空间,以及因为指令预取部件访问的字节序与定义的字节序与不一致时,引发该异常。
- Program 异常。当 E500 内核执行非法指令,或者 E500 内核在用户模式下,执行在超级模式下才能使用的指令或者 tw 指令满足条件时,E500 内核将会产生此类异常。
- DTLB miss 异常。当 E500 内核对数据空间进行读取操作,所访问的数据地址没有在 TLB0 和 TLB1 中命中时,处理器产生此异常。
- DSI 异常。在 E500 内核处于用户模式时,访问在超级模式下才能操作的内存空间时,或者数据访问的字节序与定义的字节序与不一致而引发的异常。
- System call。E500 内核执行 sc 指令时,将产生此异常。Linux PowerPC 使用这条指令实现系统调用功能。
- Decrementer 异常。由 E500 内核的 DEC 寄存器产生的定时异常,Linux PowerPC 使用此异常实现系统时钟中断。

提醒读者注意,E500 内核在进入异常处理程序中,MSR 寄存器的 EE 位将被屏蔽。因此在一般情况下,异常处理程序不会被外部中断切换。有时异常处理程序会选择合适的时机使能 EE 位,允许外部中断。此时异常处理程序需要保证,其所需的中断堆栈已经建立完毕,该程序使用的关键数据已经保存在中断堆栈中。

一般来说,E500 内核在执行异常处理程序时,不会产生新的异常。因此系统程序员在编

写异常处理程序时，需要注意不能在异常处理程序中，访问使用 TLB0 映射的物理地址空间。访问 TLB0 映射的物理地址空间时，E500 内核有可能产生 TLB Miss 异常。在基于 E500 内核的 Linux PowerPC 中，异常处理程序所使用的程序空间和数据空间都是使用 TLB1 映射的。在 603E 内核中，进入异常处理程序时，MMU 将会被自动关闭，不会产生 TLB Miss 异常。因此在 603E 内核中可以使用 TLB 映射中断及异常处理程序的地址空间。虽然如此，在基于 603E 内核的 Linux PowerPC 中，还是使用 BAT 映射中断及异常处理程序的地址空间。

6.1.1 E500 内核的中断向量

中断向量是指中断或者异常程序的入口地址。基于 603E 内核的 PowerPC 处理器，中断向量为一个固定的物理地址。在这些处理器进入中断和异常处理程序时，MMU 将会被自动关闭，因此 603E 内核可以使用固定的物理地址作为中断向量。

而 E500 内核在进入中断和异常处理程序时，不能关闭 MMU，因此不能使用物理地址作为中断向量，而应使用 IVPR 和 IVOR 寄存器保存相应的中断向量。

对于 E500 内核，中断处理程序的入口地址为虚拟地址。Linux PowerPC 使用 E500 内核的地址空间 0 保存这些虚拟地址。这是因为 E500 内核在进入中断和异常处理程序时，将使用地址空间 0 空间，有关 E500 内核地址空间的详细介绍见本书的 3.1.1 节。

基于 E500 内核的 Linux PowerPC 使用 TLB1 的 Entry0，对这些中断向量进行虚实地址映射，因为使用 TLB1 的 Entry 进行映射的虚拟地址空间将不会被替换。

如果使用 TLB0 映射这些中断向量，操作系统在进入 TLB Miss 异常处理程序时，如果再次发生中断向量所在的虚拟地址没有在 TLB0 中的情况，会再次产生 TLB Miss 异常，此时 TLB Miss 异常会出现嵌套，从而导致整个操作系统崩溃。

在 E500 内核中，使用 IVPR 和 IVORx 寄存器共同确定中断或者异常程序的入口地址。其中，IVPR 寄存器提供中断程序入口地址的第 0～15 位，IVORx 提供中断程序入口地址的第 16～27 位，而中断程序的入口地址的第 28～31 位为 0。IVORx 与异常的对应关系如表 6-1 所示。

表 6-1 E500 内核的中断向量表

IVORs	异常类型	IVORs	异常类型
IVOR0	Critical Interrupt	IVOR11	Fixed-interval timer interrupt
IVOR1	Machine Check	IVOR12	Watchdog timer interrupt
IVOR2	DSI	IVOR13	Data TLB error
IVOR3	ISI	IVOR14	Instruction TLB error
IVOR4	External Input	IVOR15	Debug
IVOR5	Alignment	IVOR16～IVOR31	保留
IVOR6	Program	IVOR32	SPE APU unavailable
IVOR7	E500 内核中不支持该异常	IVOR33	Embedded floating-point data exception
IVOR8	System Call	IVOR34	Embedded floating-point round exception
IVOR9	E500 内核中不支持该异常	IVOR35	Performance monitor
IVOR10	Decrementer	IVOR36～IVOR63	保留

本章将重点介绍 External Input 异常，即 E500 内核的外部中断。

6.1.2 外部中断处理机制

E500 内核的外部中断由三部分组成，分别为 Machine Check 异常、Critical Interrupt 异常和 External Input 异常。在 E500 内核中，Machine Check 异常采用边沿触发(edge-triggered)，而 Critical Interrupt 异常和 External Input 异常使用电平触发(level-triggered)。下文以 MPC8541 处理器为例，说明 PQIII 处理器的中断控制器 PIC。

在 PIC 中，MPC8541 处理器设置了一些寄存器位，将中断映射为电平或者边沿触发，但这些中断触发条件需要以电平的形式传递给 E500 内核。E500 内核在进入中断和中断返回时都要进行程序上下文的切换。频繁的中断将会极大影响处理器的效率。

在 E500 内核中，一个完整的中断处理流程如下：

1) E500 内核捕捉到硬件中断信号 cint＃、int＃或者 mcp。

2) E500 内核在捕捉到硬件中断信号后，需要做一些必要的准备，之后才可以进入外部中断处理程序。这些准备带来了一些外部中断处理的延时，这些延时不可避免。首先 E500 内核清除在指令完成队列 CQ 中的所有指令，除了以下三种指令：

- 在 CQ0 中的对保护区域(Guarded)的读操作指令。
- 对禁止 Cache 内存区域操作的指令。
- stwcx. 指令。

这些指令执行完毕后，E500 内核才可以进入中断模式。在清除 CQ 后，E500 中断处理模块从相应的中断向量处预取指令到 IQ 中。这些中断的预处理过程，其主要目的是为中断处理程序准备一个“干净”的空间，保证中断处理程序与被中断程序之间不相互干扰。由此可见，E500 内核进入中断处理程序之前，隐式地进行了指令同步操作。

3) E500 内核将中断程序的返回地址保存在 SRR0 中；将程序的 MSR 寄存器保存在 SRR1 中。对于一些特殊的异常，如 DSI、ISI 和 TLB Miss 等，E500 内核还会自动保留 E500 内核的其他一些寄存器。

4) E500 内核将 MSR 寄存器的 CE，ME，DE 位保留，其他位全部清零。因此 E500 内核在进行外部中断处理程序时，仍然可以被 Critical 中断，Machine check 中断和调试中断程序重入，但是不能被外部中断立即重入。在 Linux PowerPC 中，外部中断处理程序会选择合适时机使能 MSR 寄存器的 EE 位，以支持外部中断的重入。

5) MSR 中的 PR 位，EE 位，IS 和 DS 位将被清除，因此 E500 内核在超级用户模式中，运行中断处理程序时，对程序空间和数据空间的访问都要在地址空间 0 上进行。

6) E500 内核将根据 IVPR，IVOR4 寄存器确定中断向量，进行中断程序的执行。

7) 在中断处理程序执行完毕后，使用 rfi 指令进行中断返回。rfi 指令将从 SRR1 寄存器中恢复 MSR 寄存器的值，并从 SRR0 寄存器中获得程序返回地址。rfi 指令在进行程序正文切换之前还会进行指令和数据的同步，还给被中断的程序一个“干净”的空间，之后 E500 内核进行中断返回。

6.1.3 外部中断的嵌套

E500 内核提供了以下多种机制用于处理中断嵌套：

- E500 内核为不同种类的外部中断设置了屏蔽位，可以顺序地处理不同级别的外部中断。
- E500 内核为不同种类的中断(Noncritical 中断，Critical 中断，Machine Check 中断)设置了不同的寄存器组，保存相应的状态。
- E500 内核在进入外部中断处理程序时，设置中断屏蔽位防止同类外部中断的重入。

在一个操作系统中，如果外部中断处理程序可以严格执行同步规则，并且不可以被同级中断嵌套，那么中断处理程序的处理将相对简单。然而，操作系统有时需要实现中断的嵌套。

如对于外部中断的处理，E500 内核只有一个中断引脚 int #，响应不同种类的外部中断。这些外部中断源并不相同，要求的响应时间和响应级别也不相同。为此，操作系统需要实现外部中断的嵌套处理。对于系统中的一些慢速设备，如 UART、键盘等，其中断处理程序需要花费相对较长的时间。因此在一个操作系统中，这些设备的中断处理程序应该可以被其他快速设备的外部中断处理程序重入。

E500 内核执行外部中断处理程序时，会自动屏蔽所有其他的外部中断。如果外部中断处理程序支持中断的嵌套，则需要选择合适的时机将外部中断重新使能。在此之前，这些支持中断嵌套的处理程序，需要使用中断堆栈或者其他方式保存临时变量和寄存器。

Linux 系统使用中断堆栈保留这些中断现场，这样做的关键问题是如何组织中断的堆栈结构。因为操作系统在执行应用程序，系统程序和中断处理程序的过程中都可以发生中断。这些不同的中断要求系统软件必须合理地设置中断栈段。

操作系统采用两种方式进行中断栈段的处理，一种是为外部中断设立独立的堆栈，与应用程序、系统程序的栈段独立；另一种是将中断堆栈依附在操作系统的核心堆栈中。但是无论操作系统采用哪种方式的堆栈设置，都需要防止中断栈段的溢出。

对于中断频繁的应用，操作系统需要限制中断的多重嵌套。对此类应用，系统程序员可以采用软硬件结合的方法处理，如可以在硬件中设置中断阈值寄存器，将多次中断进行计数，当计数大于该阈值后，统一产生一次中断事件；在软件中断处理程序中，结合查询一次处理多个中断事件后，再重新使能外部中断屏蔽位。

为支持中断嵌套，Linux 系统的外部中断处理程序书写得较为复杂。不同的操作系统对中断嵌套的实现也不尽相同，如 Vxworks 系统为中断处理程序设置了单独的中断堆栈，而 Linux PowerPC 可以将中断堆栈依附在进程的核心堆栈中，也可以采用独立的中断堆栈模式。目前，基于 32 位 PowerPC 处理器的 Linux PowerPC，只能将中断堆栈依附在进程的核心堆栈中。

6.1.4 MPC8541 的外部中断

MPC8541 处理器采用中断控制器 PIC 管理所有的外部中断源，共支持 12 个外部中断源和 21 个内部中断源。其中，12 个外部中断源来自 MPC8541 处理器的外部引脚 IRQ[0:11]，这些引脚可以与 MPC8541 芯片设备的中断信号互联，如以太网 Phy 的中断信号，Flash 的中断信号及其一些自定义的中断信号，21 个内部中断源来自 MPC8541 内部的 SoC(System-on-Chip)，如 PCI 总线控制器中断、TSEC 中断、CPM 等一系列中断源。MPC8541 内部中断源如表 6-2 所示。

表 6-2　MPC8541 的内部中断源

内部中断号	内部中断源	描　述
0	L2 Cache	处理来自 L2 Cache 的中断
1	ECM (E500 Coherency Module)	处理来自总线共享一致性模块的中断
2	DDR SDRMA	处理来自 DDR 控制器的中断
3	LBC(Local Bus Controller)	处理来自局部总线控制器的中断
4	DMA channel 0	处理来自 DMA 控制器的中断。在 E500 内核中支持 4 个独立的 DMA 控制器。
5	DMA channel 1	
6	DMA channel 2	
7	DMA channel 3	
8	PCI 1	处理来自 PCI 总线的中断,MPC8541 支持两条独立的 PCI 总线
9	PCI 2	
13	TSEC1 transmit interrupt	处理来自 TSEC1 控制器的中断,当 TSEC1 发送、接收数据及其在发送/接收数据过程中出现错误时将产生中断。
14	TSEC1 receive interrupt	
18	TSEC1 receive/transmit interrupt	
19	TSEC2 transmit interrupt	处理来自 TSEC1 控制器的中断,当 TSEC2 发送、接收数据及其在发送/接收数据过程中出现错误时将产生中断。
20	TSEC2 receive interrupt	
24	TSEC2 receive/transmit interrupt	
26	DUART	处理来自 DUART 的中断
27	I^2C controller	处理来自 I^2C 总线控制器的中断
28	Performance monitor interrupt	处理来自 Performance monitor 的中断
29	Security	处理来自加密引擎的中断,在 MPC8541 中包含一个硬件加密引擎
30	CPM	处理来自 CPM 的中断

MPC8541 处理器设置了一系列寄存器管理内部及外部中断源,这组寄存器与 OpenPIC 和 MPIC 中断控制器基本兼容。在 MPC8541 处理器中,设置了 21 个内部中断号与 21 个内部中断源对应,12 外部中断号与外部中断源相对应。

6.1.5 MPC8541 中断控制器 PIC 的寄存器

在 MPC8541 处理器的中断控制器 PIC 中,设置了一系列寄存器。其中有些寄存器采用存储器映像方式进行寻址,可以被 MPC8541 处理器通过存储器访问指令进行访问;有些寄存器属于中断控制器 PIC 中的内部寄存器,只能被中断控制器 PIC 访问,程序员不能使用存储器指令访问这些寄存器,这些寄存器对软件透明。

MPC8541 处理器的中断控制器与 OpenPIC、MPIC 中断控制器基本兼容。OpenPIC 中断控制器由 AMD 和 Cyrix 联合提出,其设计目标是对多核处理器的外部中断进行管理,与此类似的标准还有 Intel 的 APIC 中断控制器。虽然 OpenPIC 中断控制器的设计目标主要是针对多核处理器,但是仍然可以管理单核处理器的外部中断。

MPIC 中断控制器是 IBM 为自己的 pSeries 服务器量身订制的,MPIC 中断控制器与

OpenPIC 中断控制器大同小异。在 Linux PowerPC 中，IBM 实现了基于 MPIC 中断控制器的中断处理程序，并鼓励采用 OpenPIC 中断控制器的 PowerPC 处理器(实际上，主要针对 Freescale 和 AMCC)在系统软件一级，与 IBM 的 MPIC 中断控制器兼容。

在 MPC8541 处理器中，中断控制器 PIC 的寄存器由三部分组成，分别是 Interrupt Source Configuration Registers、Per Processor Registers 和 Global Registers。

1. Interrupt Source Configuration Registers

Interrupt Source Configuration Registers 由 IIVPR0～31、IIDR0～31、EIVPR0～11，EIDR0～11 寄存器组成。其中，IIVPRx(Internal Interrupt x vector/priority Register)和 IIDRx(Internal Interrupt x destination/on Register)寄存器组与 MPC8541 的内部中断号一一对应，而 EIVPRx(External Interrupt x vector/priority Register)和 EIDRx(External Interrupt x destination/on Register)寄存器与 MPC8541 的外部中断号一一对应。

IIVPR0～31 寄存器组可读写，共由 32 个寄存器组成，其主要数据位描述如下：

MSK	A	RSV1	P	RSV2	PRIORITY	VECTOR
第 0 位	第 1 位	第 2～7 位	第 8 位	第 9～11 位	第 12～15 位	第 16～31 位

- MSK 位，读写。该位为 1 表示相应的内部中断源被屏蔽。
- A 位，只读。该位为 1 表示相应的内部中断源有效，为 0 表示无效。
- P 位，读写。该位为 1 表示相应的内部中断源为高电平有效。提醒读者注意，该位为 0 时，并不表示相应的内部中断源为低电平有效。因为在中断控制器 PIC 中，所有的内部中断源只能是高电平有效。因此该位为 0 时，表示当前内部中断源被屏蔽。
- PRIORITY 字段，读写。该字段描述内部中断源的优先级。该字段共有 4 个有效位，可以描述 15 种中断优先级。该字段的值越大表示相应的内部中断源的优先级越高，该字段为 0 表示相应的内部中断源被屏蔽。
- VECTOR 字段，读写。该字段用来存放与相应内部中断号对应的硬件中断号。该字段共有 16 个有效位，可以表示 65536 种硬件中断号。

IIDR0～31 寄存器组可读写，共由 32 个寄存器组成。主要数据位如下所示：

EP	CI	RSV	P0
第 0 位	第 1 位	第 2～30 位	第 31 位

- EP 位，读写。该位为 0，表示相应的中断源，由中断控制器 PIC 通过 int＃信号传递给处理器内核；该位为 1 表示该中断源将旁路中断控制器 PIC，而从 MPC8541 的外部引脚 IRQ_OUT＃输出。一个系统使用片外中断控制器时，可以采用这种方式，将所有的中断源统一从 IRQ_OUT＃输出，并由片外中断控制器处理，最后通过 IRQ0 引脚将中断信息送往 E500 内核。
- CI 位，读写。该位为 0，表示该中断源将由中断控制器 PIC 通过 int＃信号传递给 E500 内核；该位为 1 表示该中断源将由中断控制器 PIC 通过 cint＃信号传递给处理器内核。此位与 EP 位不能同时为 1。Linux PowerPC 一般使用 int＃信号将中断事件传递给 E500 内核。
- P0 位，只读。该位在 MPC8541 处理器中恒为 1，表示该中断源将由 CPU0 处理。在

MPC8541 处理器中只有一个 CPU,因此该位只能为 1。

MPC8541 处理器中还有两组寄存器,EIVPR0～11 和 EIDR0～11 寄存器,这两组组寄存器用来描述 MPC8541 处理器的 12 个外部中断源。

EIVPR0～11 寄存器的含义与 IIVPR0～31 寄存器组基本相同:

MSK	A	RSV1	P	S	PRIORITY	VECTOR
第 0 位	第 1 位	第 2～7 位	第 8 位	第 9 位	第 12～15 位	第 16～31 位

与 IIVPR 寄存器组相比,EIVPR0～11 寄存器组中有一个 S 位,该位为 0 时表示相应的内部中断源采用边界触发方式,为 1 时表示相应的内部中断源采用电平触发方式。这组寄存器的 P 位与 IIVPR 寄存器组中的 P 位定义不同,当 P 位为 0 时,表示相应的外部中断源为低电平有效;为 1 时表示相应的外部中断源为高电平有效。EIDR0～11 寄存器与 IIDR0～31 寄存器的定义完全相同,此处不再叙述。

在 MPC8541 处理器中,内部中断源或者外部中断源都与一个 IIVPR 或者 EIVPR 寄存器相对应。这些寄存器中的 VECTOR 字段,保存了这些内部或者外部中断源的硬件中断号。MPC8541 处理器进行中断响应周期时,会得到引发这些外部中断的硬件中断号,从而判断出是哪一个中断源引发的中断事件。

在一个实际的系统中,MPC8541 处理器可以根据外部中断源的实际情况决定是采用高电平,低电平还是采用边界触发。一般来说,在设计中多使用电平触发方式。

2. Per Processor Registers

Per Processor Registers 共由以下几个寄存器组成:

(1) IPIDR(Interprocessor Interrupt Dispatch Register)0～3 寄存器。该组寄存器用来向其他 CPU 发送 IPI 中断。在 MPC8541 处理器中,只有一个 CPU,因此该寄存器中只有第 31 位 P0 有效。对此位写 1 将引发中断。这组寄存器可以用来实现多处理器内核之间的通信。

这组寄存器也可以用来产生一些实时信号,实时信号是与 Linux 系统中的信号机制相对而言的。Linux 系统的信号机制并不能保证一个进程产生的信号能够被立即处理,这些信号只能在中断或者异常处理程序结束后处理,因此这种信号机制不能满足一些处理器系统所需要的实时性。

(2) CTPR(Processor Current Task Priority Register)寄存器。该寄存器存放当前 CPU 中运行任务的级别。当 CPU 进行中断响应时,当前中断的优先级 PRIORITY 字段被写入此寄存器中。当其他内部或者外部中断源有效时,CPU 将该中断源的优先级 PRIORITY 与 CPTR 寄存器的内容进行比较,如果该中断的优先级别较大,则中断当前 CPU 中运行任务,处理该中断请求。

CPTR 寄存器中的 TASKP 字段可以表示 15 个中断级别,当 TASKP 字段为 0xF 时将屏蔽所有外部中断。在 PowerPC 处理器中,系统程序员更习惯使用 MSR 寄存器的 EE 位屏蔽和使能外部中断。

(3) WHOAMI 寄存器。该寄存器用来存放当前 CPU 的 ID 号,对于 MPC8541 处理器,该寄存器的值为 0。对于 SMP 结构的处理器,该寄存器可以协助判断当前中断处理程序是运行在哪个 CPU 中的。

(4) IACK(Interrupt Acknowledge Register)寄存器。该寄存器用来存放当前中断源的硬件中断号,是 E500 内核外部中断处理中的一个重要的寄存器。当 MPC8541 处理器对此寄存

器进行读操作时,将启动外部中断响应周期。

(5) EOI(End of Interrupt Register)寄存器。该寄存器只有“EOI CODE”字段有效,对此字段写入 0b0000 将结束当前中断的处理。

3. Global Registers

Global Registers 共由以下几种寄存器组成:

(1) FRR(Feature Reporting Register)寄存器。该寄存器的 NIRQ 字段表示,在当前处理器中,一共有多少个中断源;NCPU 字段表示在当前处理器中一共有多少个 CPU;而 VID 字段确定当前中断控制器 PIC 的版本号。MPC8541 处理器的 NIRQ 字段为 55,表示在该处理器中共支持 54 个中断源,包括 12 个外部中断源和 32 个内部中断源(其中只有 21 个内部中断源被 MPC8541 处理器使用)。

(2) GCR(Global Configuration Register)寄存器。该寄存器的 RST 和 M 位有效。对 RST 位写 1 时,复位中断控制器 PIC,当复位完成后,该位将自动清零。

M 位为 0 时,表示当前中断控制器 PIC 被旁路,MPC8541 处理器采用中断控制器 PIC 的 Pass-Through 模式,此时 IRQ0 引脚检测到的中断信息将直接送入到 CPU。

(3) IPIVPRn(IPI Vector/Priority Registers)寄存器。该寄存器用来描述 IPI 中断的属性。该寄存器的 M 位用来使能 IPI 中断;A 位用来表示当前 IPI 中断是否有效;PRIORITY 字段用来描述 IPI 中断的级别;VECTOR 字段用来表示当前的 IPI 中断向量。其于 E500 内核的 SMP 处理器使用 PIR(Processor ID Register)寄存器识别当前进程是运行在哪个 CPU 中的。

(4) SVR(Spurious Vector Register)寄存器。中断控制器 PIC 支持 Spurious 中断,该寄存器存放 Spurious 中断使用的硬件中断号。Spurious 中断的产生原因是因为一些系统级的错误或者出于测试的考虑,在以下几种情况下 MPC8541 处理器会产生 Spurious 中断。

- int#信号有效之后,如果处理器没有及时读取 IACK 寄存器启动中断响应周期,相应的中断源失效。
- int#信号有效后,如果处理器没有及时读取 IACK 寄存器启动中断响应周期,相应中断源的 IIVPR 或者 EIVPR 寄存器的 M 位被置为无效。
- int#信号有效后,但是在处理器没有及时读取 IACK 寄存器启动中断响应周期。
- CPTR 寄存器值被一些不可预料的操作加大。
- 在没有任何中断源有效时,系统软件读取 IACK 寄存器启动中断响应周期时,PowerPC 处理器也会产生 Spurious 中断。

(5) MSGR0~3(Message Register)寄存器。在 MPC8541 的中断控制器 PIC 中,有 4 个 32 位的 Message 寄存器。使能此类中断后,向这些 Message 寄存器写入数据将产生 Messaging 中断,读取 Message 寄存器或对中断状态寄存器的相应位写 1 后,将清除此中断,该组寄存器也可以用来实现实时信号。

(6) PMMRs(Performance Monitor Mask Registers)寄存器。该组寄存器用来进行性能分析,该组寄存器是 MPC8541 处理器的自定义寄存器,OpenPIC 和 MPIC 中断控制器中并没有定义该类寄存器。

中断控制器 PIC 还有几个内部寄存器 IPR,IS 和 IRR 寄存器。此组寄存器只能由中断控制器 PIC 访问,而不能被处理器直接访问。该组寄存器将在 6.1.6 节中详细探讨。

6.1.6 MPC8541 的外部中断处理过程

MPC8541 处理器的内部或者外部中断源有效时,会将相应的中断信号传递到中断控制器 PIC 中,此时中断控制器 PIC 会进行以下操作,如图 6-1 所示。

(1) MPC8541 处理器首先使用 IPR(Interrupt Pending Register)寄存器暂存所有有效的内部及外部中断源。IPR 寄存器是中断控制器 PIC 的内部寄存器,不能被 MPC8541 处理器使用存储器指令访问。该寄存器由 54 位组成,与 MPC8541 处理器的内部和外部中断源一一对应。当 MPC8541 处理器的内部或者外部中断源有效时,IPR 寄存器的相应位被置为有效。

(2) 随后中断控制器 PIC 会对暂存在 IPR 寄存器中的中断源进行处理,并将中断优先级别最高的中断源通过 IS(Interrupt Selector)传递到 IRR(Interrupt Request Register)寄存器中。此时 IRR 寄存器保存这个优先级别最高的中断源的硬件中断号(VECTOR)和中断优先级别(PRIORITY)。

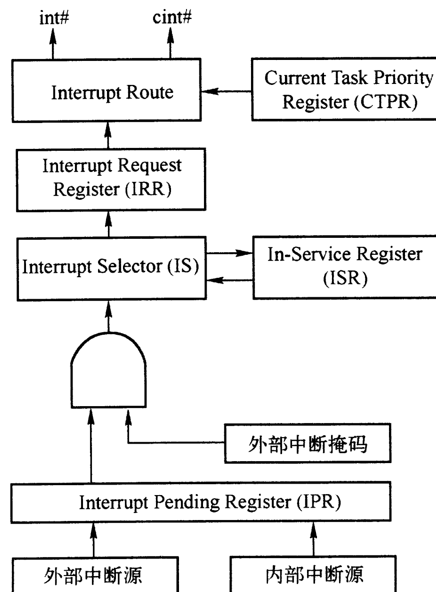


图 6-1 MPC8541 外部中断处理流程

(3) 如果在 IRR 寄存器中存放的中断源,其优先级高于 CTPR(Current Task Priority Register)和 ISR(In-Service Register)寄存器中存放的中断源,中断控制器 PIC 将使用 int#或者 cint#信号向 MPC8541 处理器发出中断请求。CTPR 寄存器存放着正在被当前中断的优先级别,该寄存器由软件进行设置。

(4) 当 int#或者 cint#信号有效时,CPU 跳转到外部中断向量,并开始执行外部中断处理程序。处理器进入中断服务程序后,将读取 IACK(Interrupt Acknowledge)寄存器,启动中断响应周期。对 IACK 寄存器的读写将使无效 int#或者 cint#信号。

(5) 在 MPC8541 处理器进行中断响应时,处理器开始真正意义上的中断处理,同时由软件更新 CTPR 寄存器为当前中断源的优先级别,并由中断控制器 PIC 在 ISR 寄存器中标记当前中断源正在被处理。MPC8541 处理器使用 ISR 寄存器,记录所有正在被处理的中断源和当前最高的中断优先级别。

(6) 在 MPC8541 中,所有的中断源都有可能引发外部中断。此时中断处理程序必须通过存放在 IACK 寄存器中的硬件中断号,判断究竟是哪一个中断源引发的外部中断,随后调用相应的中断服务例程,处理相应的中断事件。

(7) 中断事件处理完毕后,MPC8541 处理器将对 EOI(End of Interrupt Register)寄存器进行写操作以完成当前中断,并由中断控制器 PIC 自动完成对 ISR 寄存器的维护,同时使用软件更新 CTPR 寄存器为一个较低的值,以便其他中断进入。

(8) 在程序员进行外部中断处理时,可以将 CTPR 寄存器设为 0,而由 ISR 寄存器维护中断源的级别,并判断是否允许中断重入。

上述外部中断处理流程由软硬件协同完成。无论用户使用什么操作系统,E500 内核的外

部中断都需要依此进行。对于 Linux PowerPC,外部中断处理程序的设计者需要考虑整个系统的完备性,保证不同 PowerPC 处理器之间的一致。

这些要求增加了 Linux PowerPC 的外部中断处理程序的复杂性。比如在 Linux PowerPC 中,设计者需要考虑有多个中断控制器 PIC 的应用;需要考虑用户可能使用中断控制器 PIC 的 Pass-Through 功能;需要在系统的级别上,考虑外部中断的嵌套;需要兼容一些与 OpenPIC 或者 MPIC 中断控制器基本类似而不完全一致的中断控制器;需要考虑种种的错误处理。

这些种种的缜密考虑导致了 Linux PowerPC 外部中断处理的复杂性。这些缜密考虑也是普通的程序员与系统程序员之间差异所在。因为缺少这些缜密的考虑,普通程序员目前还在生产着越来越多的程序 Bug。

在 Linux 系统中,来自处理器外部或者内部的异常事件经常是系统调度、进程切换及一些系统事件的激发者。为此在 Linux PowerPC 中,设置了许多数据结构,支持外部中断处理。目前在 Linux 2.6 内核中,使用了基于 MPIC 和 OpenPIC 中断控制器的程序处理外部中断。

如果用户使用 ARCH=ppc 配置 Linux 内核,则使用基于 OpenPIC 中断控制器的程序;如果用户使用 ARCH=powerpc 配置 Linux 内核,则使用基于 MPIC 中断控制器的程序。本书将在下文详细介绍基于 MPIC 中断控制器的程序,而只对基于 OpenPIC 中断控制器的程序进行简单说明。

6.2 MPIC 中断处理程序

本书将基于 MPIC 中断控制器的中断处理程序,简称为 MPIC 中断处理程序。MPIC 中断处理程序是 Linux PowerPC 中断处理系统的核心。MPC8541 的中断控制器 PIC 与 MPIC 中断控制器基本一致。只是在 MPIC 中断处理程序中,许多代码是针对有多个中断控制器的处理器,而 MPC8541 处理器只有一个中断控制器,因此这部分代码对于 MPC8541 处理器来说,有些冗余。

在介绍 MPIC 中断处理程序之前,本章首先简单介绍 Linux PowerPC 对外部中断进行处理的流程。在本章的开始部分,曾经简单介绍了 Linux PowerPC 的外部中断处理的三大组成部分。

(1) Linux PowerPC 的外部中断系统的初始化。

(2) 设备驱动程序的中断服务例程与外部中断处理系统的挂接。

(3) 外部中断的处理过程。

在 Linux PowerPC 外部中断系统初始化时,主要完成对外部中断向量的设置,这部分内容与 MPIC 中断处理程序无关。在 Linux PowerPC 引导时,通过对 IPVR 和 IVOR4 寄存器赋值,完成外部中断向量的设置,如以下程序所示:

```
lis r4,interrupt_base@h/* IVPR only uses the high 16-bits */
mtspr SPRN_IVPR,r4
```

在这段程序中,仅将 interrupt_base 地址的高 16 位赋值给 IVPR 寄存器,这是因为 E500 内核的中断向量的入口地址为 IVPR[32-47] ‖ IVORn[49-59] ‖ 0b0000。中断向量只使用 IVPR 寄存器的高 16 地址和 IVOR 寄存器中的低 16 位。

随后,Linux PowerPC 使用宏 SET_IVOR 对与外部中断相对应的 IVPR4 寄存器进行设置,从而完成了外部中断向量的设置。

```
SET_IVOR(4,ExternalInput) ⇔
li r26,ExternalInput @l;          \
mtspr SPRN_IVOR4,r26;             \
sync
```

这段程序将 ExternalInput 函数地址的低 16 位赋值给寄存器 IVOR4。这使得当 E500 内核的外部中断来临时,外部中断处理程序可以在 IVPR 和 IVOR4 指定的地址处执行。在这段代码中,读者可以发现 Linux PowerPC 外部中断入口程序为 ExternalInput 函数。

当 E500 内核的 int# 信号有效时,MPC8541 处理器启动外部中断响应周期,通过 MPIC 中断处理程序获得外部中断源使用的硬件中断号,然后执行与硬件中断号对应的中断服务例程,最后 MPIC 中断处理程序向 EOI 寄存器写入外部中断结束命令,完成外部中断的处理。

在 Linux 系统中,外部中断源使用的中断服务例程,通过设备驱动程序挂接到外部中断处理程序中。Linux 系统出于整个系统的灵活性及可配置性的考虑,在设备驱动程序中,不直接使用硬件中断号,而是使用软件中断号将设备驱动程序中的中断服务例程,加入到 Linux 系统的外部中断处理程序中。

由此可见,Linux 系统必须在硬件中断号和软件中断号之间建立映射关系,以方便实现硬件中断到软件中断号的转换,和软件中断号到硬件中断号的转换。在 Linux PowerPC 中,系统程序员需要建立以下两种中断号映射关系:

- Linux 系统的外部中断处理程序需要将硬件中断号转换为软件中断号,然后通过软件中断号找到相应的外部中断服务例程,执行相应的外部中断处理。因此 Linux PowerPC 必须建立硬件中断号与软件中断号之间的映射。
- Linux 系统的设备驱动程序需要将软件中断号转换为硬件中断号,然后将外部中断服务例程与硬件中断号联系在一起。为此 Linux PowerPC 还要建立软件中断号与硬件中断号之间的映射。

为提高效率,Linux PowerPC 使用了两个独立的中断映射表存放这些映射关系。

Linux PowerPC 使用 MPIC 中断处理程序,完成软硬件中断号之间的转换,此外,Linux PowerPC 还使用 MPIC 中断处理程序对中断控制器中的寄存器进行读写操作。为了实现这些功能,Linux PowerPC 使用了一系列数据结构和操作函数实现 MPIC 中断处理程序。

6.2.1 MPIC 中断处理程序使用的主要数据结构

在 ./include/asm-powerpc/mpi.h 文件中,Linux PowerPC 定义了许多与 MPIC 中断处理程序有关的数据结构,其中最为重要的数据结构为 mpic 结构。在一个处理器系统中,可能有多个 MPIC 中断控制器,每一个 mpic 结构描述一个中断控制器。MPIC 中断控制器支持级联方式,一个处理器系统可以将多个 MPIC 中断控制器级联,统一处理所有外部中断源。

mpic 结构记录了 MPIC 中断控制器使用的所有寄存器、对这些寄存器操作的函数指针等其他参数,mpic 结构的各个参数如下所示。

1. device _ node 参数

```
struct mpic
{
    /* The device node of the interrupt controller */
    struct device_node *of_node;
```

Linux PowerPC 支持 Open Firmware 结构，该结构使用 device _ node 结构管理所有处理器资源，包括处理器类型、存储器大小、PCI 总线属性、所有的外部设备及中断控制器等。of _ node 参数指向 MPIC 中断控制器的 device _ node 指针，在 Linux PowerPC 引导时，该参数根据当前处理器系统的外部中断信息初始化。

在 Open Firmware 结构中还实现了一组 OP API 用来操纵 device _ node 结构。有关 Open Firmware 的详细资料请见下文。

2. irq _ host 参数

```
/* The remapper for this MPIC */
struct irq_host      *irqhost;
```

irqhost 参数保存当前 MPIC 中断处理程序使用的 irq _ host 结构指针。irq _ host 结构的主要作用是建立 MPIC 中断控制器的硬件中断号与 Linux PowerPC 软件中断号之间的联系，即为 Linux 的软件中断处理程序和硬件处理器中断号建立映射关系。

由上文所述在 Linux PowerPC 中，设备驱动程序只能使用软件中断号将自己的中断服务例程挂接到外部中断处理程序中，而在外部中断处理程序中只能读写 MPIC 中断控制器的 IACK 寄存器获得中断的硬件中断号，因此 Linux PowerPC 需要提供一种机制将硬件中断号转换为对应的软件中断号后，才能在外部中断处理程序中调用相应设备的中断服务例程，从而处理外部设备的中断请求。irq _ host 数据结构在 ./include/asm-powerpc/irq.h 文件中，其主要数据成员如下：

```
struct irq_host {
  struct list_head link;

  /* type of reverse mapping technique */
  unsigned int   revmap_type;
#define IRQ_HOST_MAP_LEGACY     0    /* legacy 8259, gets irqs 1..15 */
#define IRQ_HOST_MAP_NOMAP      1    /* no fast reverse mapping */
#define IRQ_HOST_MAP_LINEAR     2    /* linear map of interrupts */
#define IRQ_HOST_MAP_TREE       3    /* radix tree */
  union {
    struct {
      unsigned int size;
      unsigned int *revmap;
    } linear;
    struct radix_tree_root tree;
  } revmap_data;
```

```
    struct irq_host_ops  *ops;
    void      *host_data;
    irq_hw_number_t  inval_irq;
};
```

(1) link 参数。在一个处理器系统中,可能有多个中断控制器,Linux PowerPC 为每一个中断控制器提供了一个 irq_host 结构,并使用 irq_host 结构中的 link 参数将所有这些 irq_host 连接在一起,形成一个 irq_host 队列。

Linux PowerPC 使用全局指针 irq_hosts,指向这个队列的头部,全局指针 irq_hosts 在 ./arch/powerpc/kernel/irq.c 文件中定义。许多 Linux 系统的系统程序员喜欢使用 list_head 结构将一些相互关联的结构连接在一起。在 Linux 系统中,有时系统程序员对 list_head 结构的使用已经达到了泛滥的地步。实际上在 Linux 系统中,有许多结构并没有一定要使用这个结构的必要。

(2) revmap_type 参数。Linux PowerPC 为每一个 irq_host 结构定义了四种软硬件中断号之间的映射关系,并且使用 revmap_type 参数记录这种映射关系。revmap_type 参数的取值范围为 0~3,分别使用宏 IRQ_HOST_MAP_LEGACY、IRQ_HOST_MAP_NOMAP, IRQ_HOST_MAP_LINEAR 和 IRQ_HOST_MAP_TREE 表示。

- revmap_type 参数为 IRQ_HOST_MAP_LEGACY 时,当前 irq_host 结构用来描述 8259 中断控制器。8259 中断控制器是一种可编程的中断控制器,可以管理 8 个中断请求,同时,可通过多片级联实现多达 64 个的中断请求。8259 中断控制器是一个较常用的中断控制器,许多人在大学期间就已经掌握了这种中断控制器的使用。在一些处理器系统中,至今还在使用着 8259 中断控制器。
- revmap_type 参数为 IRQ_HOST_MAP_NOMAP 时,当前 irq_host 结构用来支持 IBM 的 iSeries 服务器。
- revmap_type 参数为 IRQ_HOST_MAP_TREE 时,当前 irq_host 结构采用树型结构建立处理器系统的软硬件中断号之间的映射,IBM 的 pSeries 服务器采用了这种方式建立软硬件中断号之间的映射。IBM 公司一直喜欢使用各种各样的方式,并尝试各种各样的算法。本书对 IRQ_HOST_MAP_LEGACY 和 IRQ_HOST_MAP_NOMAP 这两种中断映射方式不作介绍。
- revmap_type 参数为 IRQ_HOST_MAP_LINEAR 时,表示当前 irq_host 结构采用线型结构建立处理器系统的软硬件中断号之间的映射,Freescale 的 PowerPC 处理器采用了这种方式,该方式也是本书介绍的重点。

(3) linear 参数是 revmap_data 联合结构中的一个结构变量,linear 结构定义了两个参数,分别为 size 和 revmap 参数,size 参数和 revmap[0]共享同一个存储空间,revmap[0]没有被 irq_host 结构使用。size 参数用来确定在当前 irq_host 结构中一共支持多少个中断源,而 revmap 数组的大小由 size 参数规定。revmap 是一个整型数组,在该数组中存放着硬件中断号与软件中断号之间的映射关系。

(4) ops 参数是指向 irq_host_ops 结构的指针。在 irq_host_ops 结构中有四个操作函数:match,map,unmap 和 xlate。这些函数的主要作用是对处理器硬件中断号和操作系统软件

中断号进行映射。

在 MPIC 中断处理程序中，软件中断号和硬件中断号不是简单的一一映射关系。如果用户将一个软件中断号资源释放，之后再根据相同硬件中断号向系统申请软件中断号时，可能获得的软件中断号与原来的软件中断号不同。

使用 OpenPIC 中断处理程序时，软件中断号释放后，根据相同硬件中断号向系统申请软件中断号时，获得的软件中断号资源与之前的号码完全相同。MPIC 中断控制器的这种实现方法提高了软件中断号管理的灵活性，同时也可以充分利用软件中断号资源。

(5) host_data 参数是一个 void 类型的指针，用来保存当前 mpic 结构。

3. hc_irq、hc_ht_irq 和 hc_ipi 参数

```
  /* The "linux" controller struct */
  struct irq_chip       hc_irq;
#ifdef CONFIG_MPIC_BROKEN_U3
  struct irq_chip       hc_ht_irq;
#endif
#ifdef CONFIG_SMP
  struct irq_chip       hc_ipi;
#endif
```

hc_irq，hc_ht_irq 和 hc_ipi 是 irq_chip 结构的参数。其中 hc_ipi 参数对 SMP 结构的处理器有意义。hc_ht_irq 参数只对使用 HyperTransport 技术的处理器有意义，IBM 的 PPC970 处理器使用了这种技术，对此技术有兴趣的读者可以参考 Jay Trodden 和 Don Anderson 合著的 HyperTransport System Architecture。

irq_chip 结构描述 MPIC 中断控制器的属性。irq_chip 结构为系统软件提供了一组对中断控制器 MPIC 的寄存器，进行操作的函数集。irq_chip 结构的定义在 ./include/linux/irq.h 文件中，其主要数据成员如下：

```
struct irq_chip {
  const char *name;
  unsigned int(*startup)(unsigned int irq);
  void  (*shutdown)(unsigned int irq);
  void  (*enable)(unsigned int irq);
  void  (*disable)(unsigned int irq);

  void  (*ack)(unsigned int irq);
  void  (*mask)(unsigned int irq);
  void  (*mask_ack)(unsigned int irq);
  void  (*unmask)(unsigned int irq);
  void  (*eoi)(unsigned int irq);

  void  (*end)(unsigned int irq);
  void  (*set_affinity)(unsigned int irq,cpumask_t dest);
  int  (*retrigger)(unsigned int irq);
```

```
    int  ( * set _ type)(unsigned int irq,unsigned int flow _ type);
    int  ( * set _ wake)(unsigned int irq,unsigned int on);

    /* Currently used only by UML,might disappear one day. */
#ifdef CONFIG _ IRQ _ RELEASE _ METHOD
    void  ( * release)(unsigned int irq,void * dev _ id);
#endif

    const char * typename;
};
```

- name 参数。Linux 系统用此参数纪录在/proc/interrupts 目录下中断控制器的名称。
- startup,enable,unmask 函数用来使能对应的中断源。
- shutdown,disable 和 mask 函数用来屏蔽对应的中断源。
- ack 函数用来进行中断响应。
- mask _ ack 函数用来响应并屏蔽对应的中断源。
- eoi 函数和 end 函数结束对应中断的处理。end 函数与 eoi 函数的区别在于,当 eoi 函数执行完毕后,对于处理器而言,中断处理过程完毕。而 end 函数执行完毕后,对于操作系统而言,中断处理过程结束。
- set _ affinity 函数用来设置中断亲和属性,即该中断由 SMP 中的哪个处理器进行处理。此函数对于 Linux SMP 系统有意义。

MPIC 中断处理程序可以使用 mpic→hc _ irq 参数中的函数,对 MPIC 中断控制器的中断源寄存器组进行读写。MPIC 中断处理程序的 mpic→hc _ irq 参数中有四个重要的函数,分别是中断源屏蔽函数 mpic _ mask _ irq、中断源使能函数 mpic _ unmask _ irq、中断请求结束函数 mpic _ endirq 和中断类型设置函数 mpic _ set _ irq _ type。

4. name 参数和 flags 参数

```
const char     * name;
/* Flags */
unsigned int     flags;
```

name 参数记录 mpic 结构的名字,flags 参数主要用于 MPIC 中断处理程序的初始化。在 Linux PowerPC 中,flags 参数可以为以下数据位的组合。

- MPIC _ PRIMARY。此位为 1 时,当前 MPIC 中断控制器为主中断控制器。MPIC 中断控制器支持多个中断控制器的级联。在一个处理器系统中,只能有一个主控制器。
- MPIC _ BIG _ ENDIAN。此位为 1 时,Linux PowerPC 将使用大端方式访问当前中断控制器的寄存器。
- MPIC _ BROKEN _ U3 为 1 时,当前处理器系统使用了 HyperTransport 技术。
- MPIC _ BROKEN _ IPI 为 1 时,当前 MPIC 中断处理程序运行在 Linux SMP 系统中。
- MPIC _ WANTS _ RESET。此位为 1 表示 Linux PowerPC 在初始化 MPIC 中断处理程序时需要对 MPIC 中断控制器进行硬件复位。
- MPIC _ SPV _ EOI。此位为 1 表示处理器产生 Spurious 中断时,需要使用 eoi 函数结束

处理器的 Spurious 中断。

- MPIC _ NO _ PTHROU _ DIS。此位为 1 表示禁止 8259 的 pass-through 功能。
- MPIC _ REGSET _ STANDARD 与 MPIC _ REGSET _ TSI108。设置两个位的目的是针对一些与 MPIC 中断控制器不完全兼容的中断控制器,如 TSI108/TSI109 桥片。TSI108/109 桥片的中断控制器,其整体结构与 MPIC 中断控制器类似。但是个别寄存器的地址偏移与 MPIC 中断控制器的标准定义不同。这可能是由该芯片的设计工程师的疏忽造成的。Linux PowerPC 使用条件编译 CONFIG _ MPIC _ WEIRD 处理 TSI108/109 桥片这些非标准的 MPIC 中断控制器中断控制器。

5. 与 ISU 相关的参数

```
unsigned int        isu_size;
unsigned int        isu_shift;
unsigned int        isu_mask;
volatile u32 __iomem        *isus[MPIC_MAX_ISU];
```

Linux PowerPC 在实现 MPIC 中断控制器时,采用了 ISU(Interrupt Source Unit)技术处理分布式中断。在 ISU 中,可以将多个相似的硬件中断源划分到一个组中,然后再进行分组处理。采用这种方法的优点是可以将一些特征相似的中断源统一管理,从而在一定程度上提高了中断处理的效率。该技术也是 IBM 众多专利技术中的一员。Linux PowerPC 使用 isu _ size, isu _ shift, isu _ mask 和 isus 参数实现 ISU 技术,其含义如下所示:

- isu _ size 参数表示在一个中断组 ISU 中一共有多少个中断。
- isu _ shift 参数协助计算一个硬件中断号属于哪个 ISU 组。
- isu _ mask 参数协助计算一个硬件中断号在 ISU 组中的偏移。
- isus 数组记录 ISU 组中第一个中断向量的虚拟地址。

6. 其他参数

```
unsigned int        irq_count;
/* Number of sources */
unsigned int        num_sources;
/* Number of CPUs */
unsigned int        num_cpus;

/* The various ioremap'ed bases */
struct mpic_reg_bank        gregs;
struct mpic_reg_bank        tmregs;
struct mpic_reg_bank        cpuregs[MPIC_MAX_CPUS];

/* link */
struct mpic        *next;
};
```

在 mpic 结构中,还有一些其他参数,其描述如下:

- irq _ count 参数记录在当前 mpic 结构中有效中断源个数。

- num _ sources 参数记录在当前 mpic 结构中,共有多少个中断源。
- num _ cpus 参数记录当前 mpic 结构中,共有多少个 CPU。
- gregs 参数记录指向 MPIC 中断控制器的 Global Register 的基地址。
- tmregs 参数保存在 MPIC 中断控制器的 Global Time Register 的基地址。
- cpuregs 指针保存在 MPIC 中断控制器的 Per Processor Register 的基地址。
- next 指针指向当前系统下一个 mpic 结构。MPIC 中断控制器支持级联操作,next 指针就是为了支持这一特性而设立的。

6.2.2 MPIC 中断处理程序使用的主要变量与操作函数

Linux PowerPC 在实现 MPIC 中断处理程序时,使用了两个全局变量,分别是 mpics 和 mpic _ primary,其定义在 ./arch/powerpc/sysdev/mpic.c 文件中。

```
static struct mpic * mpics;
static struct mpic * mpic _ primary;
```

在实现 mpics 和 mpic _ primary 变量时,Linux PowerPC 使用了 static 前缀,以保证这两个变量仅在 mpic.c 文件中有效,避免了对 Linux PowerPC 全局变量的污染。

Linux PowerPC 在实现 MPIC 中断处理程序时,虽然仅使用了 C 语言,但是借用了一些面向对象程序设计的思想,这使得 Linux PowerPC 对 MPIC 中断处理程序的实现,更加模块化、更具灵活性和可移植性。

mpics 变量是一个指向 mpic 结构的指针,Linux PowerPC 使用此指针访问所有在 Linux 系统内的 mpic 结构。mpic _ primary 也是指向 mpic 结构的指针,但是此指针只用来访问当前操作系统中的 Primary MPIC 中断控制器。在支持多个 MPIC 中断控制器的处理器系统中,只有一个 MPIC 中断控制器可以作为 Primary MPIC 中断控制器。在 Linux PowerPC 中,还支持了许多操作函数和宏,用来管理和访问 MPIC 中断控制器的资源。

1. MPIC 中断控制器的寄存器读写函数

Linux PowerPC 定义了一些宏,管理和访问 MPIC 中断控制器的寄存器资源。这些宏的定义如下所示:

```
#define mpic _ read(b,r)          _ mpic _ read(mpic->flags & MPIC _ BIG _ ENDIAN,(b),(r))
#define mpic _ write(b,r,v)       _ mpic _ write(mpic->flags & MPIC _ BIG _ ENDIAN,(b),(r),(v))
#define mpic _ cpu _ read(i)      _ mpic _ cpu _ read(mpic,(i))
#define mpic _ cpu _ write(i,v)   _ mpic _ cpu _ write(mpic,(i),(v))
#define mpic _ irq _ read(s,r)    _ mpic _ irq _ read(mpic,(s),(r))
#define mpic _ irq _ write(s,r,v) _ mpic _ irq _ write(mpic,(s),(r),(v))
```

这些函数的说明如下:

- mpic _ read 和 mpic _ write 函数读写在 MPIC 中断控制器的所有寄存器。在 Linux PowerPC 中,mpic _ cpu _ read,mpic _ cpu _ write,mpic _ irq _ read 和 mpic _ irq _ write 使用_ mpic _ read 和_ mpic _ write 函数实现。
- mpic _ cpu _ read 和 mpic _ cpu _ write 函数读写在 MPIC 中断控制器中的 Per Processor Register。mpic _ irq _ read 和 mpic _ irq _ write 函数读写与硬件中断号对应的寄存器,如

IIVPR0～31,IIDR0～31 和 EIVPR0～11,EIDR0～11 寄存器。

2. hc _ irq 参数中的中断屏蔽与中断使能函数

hc _ irq 参数的中断屏蔽函数为 mpic _ mask _ irq,中断使能函数为 mpic _ unmask _ irq。其中 mpic _ mask _ irq 函数的代码如下:

```
static void mpic_mask_irq(unsigned int irq)
{
  unsigned int loops = 100000;
  struct mpic *mpic = mpic_from_irq(irq);
  unsigned int src = mpic_irq_to_hw(irq);

  mpic_irq_write(src,MPIC_INFO(IRQ_VECTOR_PRI),
      mpic_irq_read(src,MPIC_INFO(IRQ_VECTOR_PRI)) |
      MPIC_VECPRI_MASK);
  /* make sure mask gets to controller before we return to user */
  do {
    if (! loops--) {
      printk(KERN_ERR "mpic_enable_irq timeout \n");
      break;
    }
  } while(! (mpic_irq_read(src,MPIC_INFO(IRQ_VECTOR_PRI))
        & MPIC_VECPRI_MASK));
} /* End mpic_mask_irq */
```

该函数的主要作用是屏蔽相应的外部中断源。mpic _ mask _ irq 函数的输入参数为中断源对应的软件中断号 irq,该函数没有返回值,其执行流程如下:

1) 使用 mpic _ from _ irq 函数,从全局变量 irq _ desc 与 irq 参数对应的 Entry 中,获得软件中断号 irq 使用的 mpic 结构。irq _ desc 变量描述所有外部中断源的信息,该变量的详细说明见 6.3.1 节。

2) 调用 mpic _ irq _ to _ hw 函数从全局变量 irq _ map 中,获得与软件中断号对应的硬件中断号,并根据硬件中断号找到与此对应的 EIVPR 或者 IIVPR 寄存器,然后将该寄存器的 MSK 位置 1,从而屏蔽相应的中断源。全局变量 irq _ map 存放着 Linux PowerPC 中软件中断号与硬件中断号之间的映射关系,该变量的详细说明见 6.3.2 节。

3) 最后该函数使用 do _ while 循环保证相应的 MSK 位已经被成功地写到相应的寄存器 IIVPR 和 EIVPR0 中。

mpic _ unmask _ irq 函数与 mpic _ mask _ irq 函数的执行流程基本类似,只是该函数将与硬件中断号对应的 EIVPR 或者 IIVPR 寄存器中的 MSK 位置 0,从而使能 PowerPC 处理器的内部或者外部中断源。为节省篇幅,本书不再对这个函数进行详细说明。

3. hc _ irq 参数中的中断请求结束函数和中断类型设置函数

hc _ irq 参数的中断请求结束函数为 mpic _ mask _ irq,该函数的输入参数为中断源对应软件中断号 irq,其代码如下所示:

```
static void mpic_end_irq(unsigned int irq)
{
  struct mpic *mpic = mpic_from_irq(irq);
  mpic_eoi(mpic);
}
```

该程序首先根据软件中断号 irq 获得 mpic 结构，之后通过 mpic_eio 函数向 MPIC 中断控制器的 EOI 寄存器写入 0b0000 结束当前中断的处理。

hc_irq 参数的中断类型设置函数为 mpic_set_irq_type。该函数一共有两个参数，virq 和 flow_type。其中 virq 参数描述软件中断号。flow_type 参数用来用来设置中断触发类型，其可用值如下所示：

- IRQ_TYPE_NONE。该值为 flow_type 的缺省值，没有意义。
- IRQ_TYPE_EDGE_RISING。该值表示当前中断源使用上升沿触发。
- IRQ_TYPE_EDGE_FALLING。该值表示当前中断源使用下降沿触发。
- IRQ_TYPE_EDGE_BOTH。该值表示当前中断源使用上升或者下降沿触发。
- IRQ_TYPE_LEVEL_HIGH。该值表示当前中断源使用高电平触发。
- IRQ_TYPE_LEVEL_LOW。该值表示当前中断源使用低电平触发。

该函数的作用是为相应的中断源设置中断触发条件，当返回值为 0 时，表示该函数正常返回。该函数的源代码如下所示：

```
static int mpic_set_irq_type(unsigned int virq,unsigned int flow_type)
{
  struct mpic *mpic = mpic_from_irq(virq);
  unsigned int src = mpic_irq_to_hw(virq);
  struct irq_desc *desc = get_irq_desc(virq);
  unsigned int vecpri,vold,vnew;
```

- 使用 mpic_from_irq 函数获得相应的 mpic 结构。
- 使用 mpic_irq_to_hw 函数将软件中断号转换为对应的硬件中断号。
- 使用 get_irq_desc 函数根据软件中断号 virq，获得全局变量 irq_desc 中对应的 Entry。全局变量 irq_desc 中的每一个 Entry 与一个软件中断号对应。

```
  if (src >= mpic->irq_count)
    return -EINVAL;

  if (flow_type == IRQ_TYPE_NONE)
    if (mpic->senses && src < mpic->senses_count)
      flow_type = mpic->senses[src];
  if (flow_type == IRQ_TYPE_NONE)
    flow_type = IRQ_TYPE_LEVEL_LOW;

  desc->status &= ~(IRQ_TYPE_SENSE_MASK | IRQ_LEVEL);
  desc->status |= flow_type & IRQ_TYPE_SENSE_MASK;
```

```
if (flow_type & (IRQ_TYPE_LEVEL_HIGH | IRQ_TYPE_LEVEL_LOW))
  desc->status |= IRQ_LEVEL;
```

这段程序首先对 flow_type 参数进行检查，如果 flow_type 参数 IRQ_TYPE_NONE 时，将 flow_type 参数设置为 IRQ_TYPE_LEVEL_LOW，即低电平触发。

这个程序有些不足之处，这段程序应该对当前硬件中断号对应的中断源进行检查，如果当前中断源是 MPC8541 处理器的内部中断源，则应该将此参数设置为高电平触发，因为 MPC8541 处理器的内部中断源只支持高电平触发方式。

不过幸运的是，目前绝大多数系统程序员在调用该函数时，都不会使用 IRQ_TYPE_NONE 作为 flow_type 的输入参数。

```
    if (mpic_is_ht_interrupt(mpic,src))
      vecpri = MPIC_VECPRI_POLARITY_POSITIVE |
        MPIC_VECPRI_SENSE_EDGE;
    else
      vecpri = mpic_type_to_vecpri(mpic,flow_type);

    vold = mpic_irq_read(src,MPIC_INFO(IRQ_VECTOR_PRI));
    vnew = vold & ~(MPIC_INFO(VECPRI_POLARITY_MASK) |
        MPIC_INFO(VECPRI_SENSE_MASK));
    vnew |= vecpri;
    if (vold != vnew)
      mpic_irq_write(src,MPIC_INFO(IRQ_VECTOR_PRI),vnew);

    return 0;
} /* End mpic_set_irq_type */
```

上述程序的主要作用是初始化 MPIC 中断控制器中的 IIVPRx 或者 EIVPRx 寄存器。

- 首先如果处理器不支持 HyperTransport 结构，则需要使用 mpic_type_to_vecpri 函数将 flow_type 参数进行转换然后，再存入 vecpri 变量中。mpic_type_to_vecpri 函数的实现十分简单，本书不对此函数进行介绍，但是建议读者阅读该函数的源代码。该函数的源代码在 ./arch/powerpc/sysdev/mpic.c 文件中。
- 之后，这段程序将 vecpri 变量写到 MPC8541 的 IIVPR0～31 或者 EIVPR0～11 寄存器中，在外部中断程序初始化时，需要调用该函数初始化所有的中断源。

4. mpic_assign_isu 函数

```
void __init mpic_assign_isu(struct mpic *mpic,unsigned int isu_num,
        phys_addr_t paddr)
{
  unsigned int isu_first = isu_num * mpic->isu_size;

  BUG_ON(isu_num >= MPIC_MAX_ISU);
```

```
    mpic_map(mpic,paddr,&mpic->isus[isu_num],0,
      MPIC_INFO(IRQ_STRIDE) * mpic->isu_size);
    if ((isu_first + mpic->isu_size) > mpic->num_sources)
      mpic->num_sources = isu_first + mpic->isu_size;
}
```

mpic_assign_isu 函数对 MPIC 中断控制器的 Interrupt Source Configuration Registers 寄存器进行虚实地址转换,并将得到的虚拟地址存入 mpic 结构的 isus 参数中,之后更新 mpic 结构中的 num_sources 参数。

5. irq_alloc_host 函数

irq_alloc_host 函数用来分配 mpic 结构中的 irq_host 参数,该函数成功结束后将返回已分配的 irq_host 结构指针。

irq_alloc_host 函数的定义在 ./arch/powerpc/kernel/irq.c 文件中,该函数将建立硬件中断号与软件中断号之间的映射表 revmap,其源代码的详细说明如下:

```
struct irq_host * irq_alloc_host(unsigned int revmap_type,
            unsigned int revmap_arg,
            struct irq_host_ops * ops,
            irq_hw_number_t inval_irq)
{
```

该函数一共有四个输入参数,如下:

- revmap_type 参数可以为 IRQ_HOST_MAP_LEGACY、IRQ_HOST_MAP_NOMAP,IRQ_HOST_MAP_LINEAR 或者 IRQ_HOST_MAP_TREE,其含义如 6.2.1 节所示。对于 MPC8541 处理器,该参数的值为 IRQ_HOST_MAP_LINEAR。
- revmap_arg 参数记录在当前系统中,一共有多少个硬件中断号。
- ops 参数记录当前 irq_host 结构的操作函数指针。
- inval_irq 参数记录 Spurious 中断使用的软件中断号,Spurious 中断的详细说明见 6.1.2 节。

```
struct irq_host * host;
unsigned int size = sizeof(struct irq_host);
unsigned int i;
unsigned int * rmap;
unsigned long flags;

/* Allocate structure and revmap table if using linear mapping */
if (revmap_type == IRQ_HOST_MAP_LINEAR)
  size += revmap_arg * sizeof(unsigned int);
```

irq_alloc_host 函数首先计算当前 irq_host 结构所使用的物理内存大小。如果当前 irq_host 采用线型结构(即 revmap_type 参数为 IRQ_HOST_MAP_LINEAR)时,irq_host 结构和 revmap 表使用同一段物理地址空间。

如果 revmap 表中的对应位是 IRQ _ NONE,则表示该硬件中断号没有与 Linux PowerPC 中的软件中断号建立映射关系;若不是 IRQ _ NONE,则表示该硬件中断号已经与 Linux PowerPC 中的软件中断号建立映射关系。

在 revmap 表中存放着与硬件中断号对应的软件中断号,该表也被称作反向中断映射表(Reverse Map)。Linux PowerPC 使用 irq _ host 的 revmap _ data.linear.revmap 参数保存这个反向中断映射表。当程序员使用 IRQ _ HOST _ MAP _ LINEAR 进行硬软件中断映射时,反向中断映射表 revmap 将存放在当前的 irq _ host 结构之后,其结构如图 6-2 所示。

反向中断映射表是与中断映射表相对而言的。在 Linux PowerPC 中,还存在另外一张中断映射表。Linux PowerPC 使用全局数组 irq _ map 保存这张表,该全局数组用来保存软件中断号与硬件中断号之间的映射关系,它在 ./arch/powerpc/kernel/irq.c 文件中。下文将详细讨论这两张表的使用方法。



图 6-2　irq _ host 结构和中断映射表

```
if (mem_init_done)
  host = kzalloc(size,GFP_KERNEL);
else {
  host = alloc_bootmem(size);
  if (host)
    memset(host,0,size);
}
if (host == NULL)
  return NULL;

/* Fill structure */
host->revmap_type = revmap_type;
host->inval_irq = inval_irq;
host->ops = ops;
```

这段程序使用 kzalloc 函数或者 alloc _ bootmem 函数为 irq _ host 结构和中断映射表申请物理内存空间。当 mem _ init _ done 变量为 1 时,表示 mem _ init 函数已经执行完毕,SLAB 分配器已经被初始化,此时可以使用 kzalloc 函数在 SLAB 分配器中分配物理内存;否者只能使用 alloc _ bootmem 函数在 Linux 的 Boot Memory 中申请物理内存空间。SLAB 分配器和 Boot Memory 的详细描述见第 7 章。

无论使用 kzalloc 还是 alloc _ bootmem 函数,irq _ host 结构使用的物理内存大小都为 irq _ host 结构和中断映射表 revmap _ type 所需空间之和。irq _ host 结构分配完毕后,将 revmap _ type,inval _ irq 和 ops 参数初始化。

```
  spin_lock_irqsave(&irq_big_lock,flags);
  if (revmap_type == IRQ_HOST_MAP_LEGACY) {
...
  }
```

```
        list_add(&host->link,&irq_hosts);
        spin_unlock_irqrestore(&irq_big_lock,flags);
```

这段程序将当前的 irq_host 结构加入到 irq_hosts 链表中。由于 irq_host 结构是一个临界资源,因此需要使用自旋锁将 irq_hosts 链表锁定。

```
        switch(revmap_type) {
    ...
        case IRQ_HOST_MAP_LINEAR:
            rmap = (unsigned int *)(host + 1);
            for (i = 0;i < revmap_arg;i++)
                rmap[i] = IRQ_NONE;
            host->revmap_data.linear.size = revmap_arg;
            smp_wmb();
            host->revmap_data.linear.revmap = rmap;
            break;
        default:
            break;
        }
        pr_debug("irq:Allocated host of type %d @0x%p\n",revmap_type,host);
        return host;
    } /* End irq_alloc_host */
```

该段程序的执行流程如下:

- 当 revmap_type 参数为 IRQ_HOST_MAP_LINEAR 时,将 rmap 指向 irq_host 结构之后的中断映射表 revmap。
- 将中断映射表 revmap 中的所有 Entry 置为 IRQ_NONE。
- 初始化 revmap_data.linear 的 revmap_arg 和 revmap 参数。
- 最后将 irq_alloc_host 函数返回。至此反向中断映射表中所有 Entry 被初始化完毕。

6. mpic_alloc 函数

在 Linux PowerPC 中,一个处理器系统的每个 MPIC 中断控制器都有自己的 mpic 结构。mpic_alloc 函数的主要作用是为 mpic 结构分配空间,之后初始化这个结构。mpic_alloc 函数的返回值指向新建立的 mpic 结构,该函数一共有以下 6 个输入参数:

```
    struct mpic * __init mpic_alloc(struct device_node * node,
                phys_addr_t phys_addr,
                unsigned int flags,
                unsigned int isu_size,
                unsigned int irq_count,
                const char * name)
    {
```

- node 参数。该参数存放当前 MPIC 中断控制器使用的 Device Node。Device Node 在 Linux PowerPC 初始化时创建。

● phys_addr 参数。该参数存放当前 MPIC 中断控制器使用的基地址。

● flags、isu_size 和 irq_count 参数。flags 参数存放 mpic 结构的 flags 参数，mpic_alloc 函数将对 flags 参数进行分析。有关 flags、isu_size 和 irq_count 参数的详细信息请参考 6.2.1 节。

```
struct mpic * mpic;
u32     reg;
const char * vers;
int  i;
u64     paddr = phys_addr;

mpic = alloc_bootmem(sizeof(struct mpic));
if (mpic == NULL)
  return NULL;

memset(mpic,0,sizeof(struct mpic));
mpic->name = name;
mpic->of_node = node ? of_node_get(node) :NULL;
```

mpic_alloc 函数首先使用 alloc_bootmem 函数，在 Linux 系统的 Boot Memory 中，申请 mpic 结构所使用的物理地址空间。在 Linux PowerPC 中，mpic_alloc 函数执行时，mem_init 函数还没有被执行，因此 mpic 结构只能使用 Boot Memory 中的物理地址空间。有关 Boot Memeory 的详细信息请参考 7.2.4 节。随后 mpic_alloc 函数将 mpic 结构初始化为 0，并初始化 mpic 结构中的 name 参数和 of_node 参数。

```
mpic->irqhost = irq_alloc_host(IRQ_HOST_MAP_LINEAR,256,
            &mpic_host_ops,
            MPIC_VEC_SPURRIOUS);
if (mpic->irqhost == NULL) {
  of_node_put(node);
  return NULL;
}
```

mpic_alloc 函数调用 irq_alloc_host 函数，初始化 mpic 结构的 irq_host 参数。该函数的返回值指向新建立的 irqhost 结构。irq_alloc_host 函数共有四个参数，分别为 revmap_type、revmap_arg、ops 和 inval_irq。

对于 E500 内核，mpic_alloc 函数在调用 irq_alloc_host 函数时，revmap_type 参数为 IRQ_HOST_MAP_LINEAR，表示当前 irq_host 结构采用线型结构建立处理器系统的软硬件中断号之间的映射；revmap_arg 参数为 256，表示反向中断映射表的大小为 256 个字节；ops 参数为全局变量 mpic_host_ops，用来指向 irq_host 结构使用的 irq_host_ops 函数集。

```
mpic->irqhost->host_data = mpic;
mpic->hc_irq = mpic_irq_chip;
mpic->hc_irq.typename = name;
```

```
    if (flags & MPIC_PRIMARY)
        mpic->hc_irq.set_affinity = mpic_set_affinity;

    mpic->flags = flags;
    mpic->isu_size = isu_size;
    mpic->irq_count = irq_count;
    mpic->num_sources = 0; /* so far */
```

这段代码的执行顺序如下：

1）将 mpic->hc_irq 赋值为 mpic_irq_chip。hc_irq 为 irq_chip 类型的数据，由上文可知，irq_chip 结构主要用来描述硬件中断控制器的属性。在 mpic_irq_chip 结构中，使能中断、屏蔽中断、响应中断与设置中断类型的操作函数分别为 mpic_mask_irq、mpic_unmask_irq、mpic_end_irq 和 mpic_set_irq_type。

2）判断当前分配的 mpic 结构的 flags 参数是否有 MPIC_PRIMARY 标志，如果有则将其 hc_irq 参数的 set_affinity 函数赋值。Linux PowerPC 规定只有 Primary MPIC 控制器可以为中断源设置亲和性属性。在 Linux SMP 中，可以指定某个中断源由与它亲和的 CPU 进行处理。

3）最后将 mpic 结构的 flags，isu_size，isu_count 和 num_sources 赋值。在 MPC85XXCDS 处理器系统中，flags 的值为 MPIC_PRIMARY | MPIC_WANTS_RESET | MPIC_BIG_ENDIAN，其含义见上文；isu_size 的值为 4 表示在 ISU 中的中断个数为 4；irq_count 和 num_source 参数被初始化为 0。

```
    ...
    /* Map the global registers */
    mpic_map(mpic, paddr, &mpic->gregs, MPIC_INFO(GREG_BASE), 0x1000);
    mpic_map(mpic, paddr, &mpic->tmregs, MPIC_INFO(TIMER_BASE), 0x1000);
```

这段程序将 MPIC 中断控制器的 Global Registers 和 Global Timer Registers 进行虚实地址映射，并使用 gregs 和 tmregs 保存这些寄存器组的虚拟地址。

```
    /* Reset */
    if (flags & MPIC_WANTS_RESET) {
        mpic_write(mpic->gregs, MPIC_INFO(GREG_GLOBAL_CONF_0),
                mpic_read(mpic->gregs, MPIC_INFO(GREG_GLOBAL_CONF_0))
                | MPIC_GREG_GCONF_RESET);
        while( mpic_read(mpic->gregs, MPIC_INFO(GREG_GLOBAL_CONF_0))
                & MPIC_GREG_GCONF_RESET)
            mb();
    }
```

这段程序根据 flag 参数，确定是否需要对 MPIC 中断控制器进行复位。如果需要复位，则向 MPIC 中断控制器中的 Global Configuration Registers 寄存器的 R 位写 1，当 MPIC 中断控制器复位完成后，此位将被自动清零。

```
reg = mpic_read(mpic->gregs,MPIC_INFO(GREG_FEATURE_0));
mpic->num_cpus = ((reg & MPIC_GREG_FEATURE_LAST_CPU_MASK)
      >> MPIC_GREG_FEATURE_LAST_CPU_SHIFT) + 1;
if (isu_size == 0)
  mpic->num_sources = ((reg & MPIC_GREG_FEATURE_LAST_SRC_MASK)
      >> MPIC_GREG_FEATURE_LAST_SRC_SHIFT) + 1;
```

这段程序读取 MPIC 中断控制器的 Feature Reporting Register 0 寄存器，获得当前中断控制器的版本号，支持的 CPU 数量和支持的外部中断数量，之后对 mpic 结构的 num_cpus 和 num_sources 参数赋值。

```
/* Map the per-CPU registers */
for (i = 0;i < mpic->num_cpus;i++) {
  mpic_map(mpic,paddr,&mpic->cpuregs[i],
    MPIC_INFO(CPU_BASE) + i * MPIC_INFO(CPU_STRIDE),
    0x1000);
}

/* Initialize main ISU if none provided */
if (mpic->isu_size == 0) {
  mpic->isu_size = mpic->num_sources;
  mpic_map(mpic,paddr,&mpic->isus[0],
    MPIC_INFO(IRQ_BASE),MPIC_INFO(IRQ_STRIDE) * mpic->isu_size);
}
mpic->isu_shift = 1 + __ilog2(mpic->isu_size - 1);
mpic->isu_mask = (1 << mpic->isu_shift) - 1;
```

这段程序首先使用 ioremap 函数，将 MPIC 中断控制器中的 Per Processor Registers 寄存器进行虚拟地址映射。在 MPC8541CDS 处理器系统中一共支持 1 个 CPU，因此只需要对一组 Per Processor Registers 进行虚拟地址映射。

如果 isu_size 为 0，则表示不使用 MPIC 中断控制器的 ISU 分组功能，此时所有的中断源将独立设置，并将 MPIC 中断控制器中的 Interrupt Source Configuration Registers 进行虚拟地址映射。在 MPC8541CDS 处理器系统中，isu_size 为 4 表示使用 ISU 分组功能。

需要提醒读者注意：在 MPC8541 处理器中，PIC 中断控制器实际上并不支持中断的 ISU 分组功能。但是考虑到与 IBM 的 MPIC 中断控制器兼容，Freescale 的工程师还是将 isu_size 设为 4，以便使用 mpic_assign_isu 函数进一步设置每一个外部中断源。对于 MPC8541 处理器，isu_size 为 0 或者 32 都是可以的，将 isu_size 的值设置为 32 后，以后的源代码还会更精炼些。

之后这段程序将 isu_shift 和 isu_mask 赋值。对于 MPC8541CDS 处理器系统，isue_size 参数为 4。因此 isu_shift 参数为 2，而 isu_mask 参数为 0x00000003。

```
...
  mpic->next = mpics;
  mpics = mpic;
```

```
    if (flags & MPIC_PRIMARY) {
      mpic_primary = mpic;
      irq_set_default_host(mpic->irqhost);
    }
    return mpic;
}
```

这段程序对全局变量 mpics 和 mpic_primary 进行赋值，之后 Linux 系统使用全局变量 mpic_primary 和 mpics 访问当前处理器系统的 mpic 结构。最后 mpic_alloc 函数将获得的 mpic 变量作为返回值。

6.2.3 使用 MPIC 中断控制程序对中断系统初始化

下文以 Freescale 的 MPC8541CDS 处理器系统为例，说明 Linux PowerPC 外部中断系统的初始化。如果用户在编译 Linux 内核时使用 arch=ppc 时系统将使用基于 OpenPIC 中断控制器的源代码进行中断系统的初始化，如果用户使用 arch=powerpc 编译内核，系统将使用基于 MPIC 中断控制器的源代码初始化外部中断系统。

与 MPIC 中断处理程序相比，OpenPIC 中断处理程序的历史更加悠久。但是对于 Linux 系统，有时后加入的代码更加成熟，也更加稳定些。在 Linux PowerPC 中，MPIC 中断处理程序代表着未来的方向。也许在不久的将来，OpenPIC 中断处理程序将会逐渐被淘汰。因此本文将主要讲述如何使用 MPIC 中断处理程序，初始化 Linux PowePC 外部中断系统。

Linux 系统使用 init_IRQ 函数对外部中断系统进行初始化。init_IRQ 函数分别调用两个函数 ppc_md.init_IRQ 和 irq_ctx_init。

```
void __init init_IRQ(void)
{
  ppc_md.init_IRQ();
#ifdef CONFIG_PPC64
  irq_ctx_init();
#endif
}
```

ppc_md.init_IRQ 函数与支持的处理器系统有直接的关系。在 MPC8541CDS 处理器系统中，ppc_md.init_IRQ 的值等于 mpc85xx_cds_pic_init。有关 ppc_md 结构初始化赋值的详细介绍见 8.2.5 节。

irq_ctx_init 函数用来设置独立的中断堆栈。目前 Linux PowerPC 中，只有一些 64 位的 PowerPC 处理器采用独立的中断堆栈。即便如此，使用条件编译参数 CONFIG_PPC64 判断是否执行 irq_ctx_init 函数仍然不妥当。因为也许以后，Linux PowerPC 会在 32 位处理器中使用独立的中断堆栈。

有的实时操作系统，如 VxWorks，使用独立的中断堆栈处理外部中断。采用这种方案的优点是可以将中断堆栈与其他进程的堆栈隔离，有利于堆栈空间的维护。而且在 CPU 进入中断处理程序时，不用建立中断处理程序使用的堆栈空间。

采用独立的中断堆栈在一定程度上,可以缩短中断处理延时。但是使用这种方法在维护中断堆栈时,操作系统在进入和退出中断处理程序时,需要进行中断堆栈空间的切换,从而又增加了一些中断处理延时。因此采用独立的中断堆栈也许并不能缩短中断处理延时。目前尚未发现采用独立的中断堆栈可以缩短中断处理延时的权威性文章。不过可以肯定的是,采用这种方法有利于维护中断程序的堆栈空间。

在 MPC8541CDS 处理器系统中,Linux PowerPC 使用 mpc85xx_cds_pic_init 函数初始化外部中断处理系统。

mpc85xx_cds_pic_init 函数在 ./arch/powerpc/platforms/85xx/mpc85xx_cds.c 文件中。该函数进行了以下初始化步骤:

1) 使用 Open Fireware 机制,获得 MPC8541CDS 处理器系统的中断控制器信息。

2) 调用 mpic_alloc 函数,为 mpic 结构分配内存空间并进行基本的初始化。

3) 使用 mpic_assign_isu 函数,对 MPC8541 处理器的中断控制器与中断源相关的物理地址进行虚实地址映射。

4) 使用 mpic_init 函数对 MPIC 中断控制器进行初始化。

1. 获得 MPIC 中断控制器的信息

mpc85xx_cds_pic_init 函数没有输入参数,也没有返回值。该函数的源代码如下:

```
void __init mpc85xx_cds_pic_init(void)
{
  struct mpic *mpic;
  struct resource r;
  struct device_node *np = NULL;
#ifdef CONFIG_PPC_I8259
  struct device_node *cascade_node = NULL;
  int cascade_irq;
#endif

  np = of_find_node_by_type(np,"open-pic");

  if (np == NULL) {
    printk(KERN_ERR "Could not find open-pic node\n");
    return;
  }
  if (of_address_to_resource(np,0,&r)) {
    printk(KERN_ERR "Failed to map mpic register space\n");
    of_node_put(np);
    return;
  }
```

mpc85xx_cds_pic_init 函数首先调用 of_find_node_by_type 函数获得 MPC8541CDS 处理器系统的有关中断控制器的 device_node 结构,并存放到变量 np 中。MPC8541CDS 处理器系统中断控制器信息存放在 ./arch/powerpc/boot/dts/mpc8541cds.dts 文件中,该文件包含

了 MPC8541CDS 处理器系统上的所有资源信息。在 Linux PowerPC 中，使用 Open Fireware 构架管理所有的硬件资源，并为每一个硬件资源设立一个 device _ node 结构。

mpc85xx _ cds _ pic _ init 函数使用 device _ node 结构，调用 of _ address _ to _ resource 函数，进一步获得对这些硬件资源进行描述的信息，并将这些信息保存在 resource 结构中。resource 结构的信息主要包括该中断控制器使用内存的开始地址和结束地址。

Open Fireware 构架包含一组操作函数集 OP API，包含了一系列以 op _开头的函数。在 mpc85xx _ cds _ pic _ init 函数中，使用的 of _ find _ node _ by _ type 和 of _ address _ to _ resource 函数属于这个函数集。

2. 初始化 mpic _ alloc 结构

Linux PowerPC 可以支持多个 MPIC 中断控制器，其中每一个中断控制器唯一对应一个 mpic 结构。Linux PowerPC 使用 mpic _ alloc 函数初始化 mpic 结构。mpc85xx _ cds _ pic _ init 函数使用以下语句调用 mpic _ alloc 函数：

```
mpic = mpic_alloc(np,r.start,
    MPIC_PRIMARY | MPIC_WANTS_RESET | MPIC_BIG_ENDIAN,
    4,0," OpenPIC  ");
BUG_ON(mpic == NULL);

/* Return the mpic node */
of_node_put(np);
```

mpc85XX _ cds _ pic _ init 函数在调用 mpic _ alloc 函数时，irq _ count 参数为 0，表示在当前处理器系统中有效的中断源与 mpic 结构中的中断数量相等；flags 参数为 MPIC _ PRIMARY | MPIC _ WANTS _ RESET | MPIC _ BIG _ ENDIAN，表示 mpic _ alloc 函数即将为 Primary MPIC 中断控制器分配 mpic 结构，采用大端方式访问该控制器内部的寄存器，同时需要对中断控制器进行复位。

3. mpic _ assign _ isu 函数

mpc85xx _ cds _ pic _ init 函数在初始化 mpic 结构后，使用 mpic _ assign _ isu 函数对 MPIC 中断控制器的 Interrupt Source Configuration Registers 寄存器进行虚实地址转换。

```
mpic_assign_isu(mpic,0,r.start + 0x10200);
mpic_assign_isu(mpic,1,r.start + 0x10280);
mpic_assign_isu(mpic,2,r.start + 0x10300);
mpic_assign_isu(mpic,3,r.start + 0x10380);
mpic_assign_isu(mpic,4,r.start + 0x10400);
mpic_assign_isu(mpic,5,r.start + 0x10480);
mpic_assign_isu(mpic,6,r.start + 0x10500);
mpic_assign_isu(mpic,7,r.start + 0x10580);
```

在上述代码中，mpc85xx _ cds _ pic _ init 函数调用 mpic _ assign _ isu 函数，对 MPC8541 处理器中的 IIVPR0～31 和 IIDR0～31 寄存器组进行虚实地址转换。mpic _ assign _ isu 函数调用 ioremap 函数，将 MPIC 中断控制器中的 IIVPR 和 IIDR 寄存器组映射到 Linux 系统的虚拟地址空间中。

在 MPC8541 处理器中，一共有 32 对 IIVPR 和 IIDR。Linux PowerPC 将这 32 对寄存器每 4 个分为一组，一共使用 8 组 ISU 来描述这些寄存器。其中第 1 组 ISU 与 IIVPR0～3 和 IIDR0～3 寄存器对应，而第 8 组 ISU 与 IIVPR28～31 和 IIDR28～31 寄存器对应。

IIVPR0 寄存器相对中断控制器 PIC 基地址的偏移为 0x10200，而 IIVPR28 寄存器相对中断控制器 PIC 基地址偏移为 0x10580。MPC8541 处理器并不支持 ISU 结构，因此不将这 32 对寄存器进行分组也是可以的。

```
/* Used only for 8548 so far,but no harm in
 * allocating them for everyone */
mpic_assign_isu(mpic,8,r.start + 0x10600);
mpic_assign_isu(mpic,9,r.start + 0x10680);
mpic_assign_isu(mpic,10,r.start + 0x10700);
mpic_assign_isu(mpic,11,r.start + 0x10780);
```

MPC8548 处理器一共有 48 组 IIVPR0～47 和 IIDR0～47 寄存器，因此以上代码的主要目的是对 IIVPR32～47 和 IIDR0～47 寄存器进行虚实地址转换。对于 MPC8541 处理器，以上代码是冗余的。

```
/* External Interrupts */
mpic_assign_isu(mpic,12,r.start + 0x10000);
mpic_assign_isu(mpic,13,r.start + 0x10080);
mpic_assign_isu(mpic,14,r.start + 0x10100);
```

mpc85xx_cds_pic_init 函数继续调用 mpic_assign_isu 函数对 MPC8541 处理器中的 EIVPR0～11 和 EIDR0～11 寄存器组进行虚实转换。在 MPC8541 处理器中，一共有 12 对 EIVPR 和 EIDR 寄存器，因此在 Linux PowerPC 中一共使用 3 组 ISU。

其中第 13 组 ISU 与 EIVPR0～3 和 EIDR0～3 寄存器对应，而第 15 组 ISU 与 EIVPR8～11 和 EIDR8～11 寄存器对应。EIVPR0 寄存器相对中断控制器 PIC 基地址偏移为 0x10000，而 IIVPR8 寄存器相对中断控制器 PIC 基地址偏移为 0x10100。

至此，Linux PowerPC 将 MPC8541 处理器中所有与 MPIC 中断控制器相关的寄存器都映射到了虚拟地址空间。

4. MPIC 中断控制器的初始化

在 mpic_assign_isu 函数为 MPIC 中断控制器中的 Interrupt Source Configuration Registers 进行虚拟地址映射后，mpc85xx_cds_pic_init 函数使用 mpic_init 函数对 MPIC 中断控制器进行最后的初始化。

```
    mpic_init(mpic);
} /* End mpc85xx_cds_pic_init */
```

在 6.2.2 节中，mpic_alloc 函数对 mpic 结构进行了许多初始化处理，但是那些初始化操作仅仅是软件意义上对 MPIC 中断控制器的初始化。

mpic_init 函数对 MPIC 中断控制器中的寄存器进行操作，完成硬件意义上对 MPIC 中断控制器的初始化。mpic_init 函数只有一个输入参数 mpic，该参数用来指向 mpic 结构。该函数没有返回值，其详细描述如下所示：

```
/* mpic_init 函数源代码片段 1 */
void __init mpic_init(struct mpic *mpic)
{
  int i;

  BUG_ON(mpic->num_sources == 0);
  WARN_ON(mpic->num_sources > MPIC_VEC_IPI_0);
  /* Sanitize source count */
  if (mpic->num_sources > MPIC_VEC_IPI_0)
    mpic->num_sources = MPIC_VEC_IPI_0;

  printk(KERN_INFO "mpic: Initializing for %d sources\n",mpic->num_sources);

  /* Set current processor priority to max */
  mpic_cpu_write(MPIC_INFO(CPU_CURRENT_TASK_PRI),0xf);

  /* Initialize timers:just disable them all */
  for (i = 0;i < 4;i++) {
    mpic_write(mpic->tmregs,
        i * MPIC_INFO(TIMER_STRIDE) +
        MPIC_INFO(TIMER_DESTINATION),0);
    mpic_write(mpic->tmregs,
        i * MPIC_INFO(TIMER_STRIDE) +
        MPIC_INFO(TIMER_VECTOR_PRI),
        MPIC_VECPRI_MASK |
        (MPIC_VEC_TIMER_0 + i));
  }
```

mpic_init 函数首先对 num_sources 参数进行边界检查。之后该函数将 Per Processor Registers 中的 Current Task Priority Registers 赋值为 0xf。MPIC 中断控制器可以为每一个中断设置优先权，当中断的优先权不大于 Current Task Priority Registers 寄存器时，此中断不能通过 int# 或者 cint# 信号传递给 CPU。

Current Task Priority Registers 寄存器的最大值为 0xf，此时将禁止所有中断，因为外部中断优先权的最大值为 0xf。在 PowerPC 体系中一般使用 MSR 寄存器中的 EE 位禁止或者打开中断。之后，该函数对 MPIC 中断控制器中的 4 个 Global Time Register 进行初始化。

```
/* mpic_init 函数源代码片段 2 */
...
  for (i = 0;i < mpic->num_sources;i++) {
    /* start with vector = source number,and masked */
    u32 vecpri = MPIC_VECPRI_MASK | i |
      (8 << MPIC_VECPRI_PRIORITY_SHIFT);
```

```
        /* init hw */
        mpic_irq_write(i,MPIC_INFO(IRQ_VECTOR_PRI),vecpri);
        mpic_irq_write(i,MPIC_INFO(IRQ_DESTINATION),
                1 << hard_smp_processor_id());
    }
```

这段程序将 MPIC 中断控制器中的 Vector/Priority 寄存器使用 mpic_irq_write 函数进行初始化。将其中的"vecpri = MPIC_VECPRI_MASK| i |(8 << MPIC_VECPRI_PRIORITY_SHIFT)"改为"vecpri = MPIC_VECPRI_MASK|(8 << MPIC_VECPRI_PRIORITY_SHIFT)|i"逻辑上更好些,因为 i 字段实际上是 Vector/Priority 寄存器的最后一个字段。

这段程序是 mpci_init 函数最重要的部分。在这个 for 循环中,使用 mpic_irq_write 函数对 MPIC 中断控制器的所有内部中断和外部中断 Vector/Priority 寄存器的字段进行赋值,其中 VEC 字段保存对应寄存器的硬件中断号。当发生外部中断时,外部中断处理程序将读取中断响应寄存器 IACK 获得此硬件中断号,并通过此硬件中断号判断中断类型。

mpic_irq_write 函数的源代码如下所示:

```
mpic_irq_write(i,MPIC_INFO(IRQ_VECTOR_PRI),vecpri)  ⇔
_mpic_irq_write(mpic,i,MPIC_IRQ_VECTOR_PRI,vecpri)
{
  unsigned int isu = i >> mpic->isu_shift;
  unsigned int idx = i & mpic->isu_mask;
  _mpic_write(mpic->flags & MPIC_BIG_ENDIAN,mpic->isus[isu],
      MPIC_IRQ_VECTOR_PRI + (idx * MPIC_INFO(IRQ_STRIDE)),vecpri);
}
```

在 mpic_init 的源代码中,可以发现在 for 循环的 mpic_irq_write 函数将 Vector/Priority 寄存器中的 MSK 位初始化为 1,屏蔽该中断源;将 P 位和 S 位初始化为 0;将 PRIORITY 字段初始化为 8;将 VECTOR 字段设置为 i。

由上文的 mpic_assign_isu 函数可知,mpic->isus 数组的第 0~7 项与 MPC8541 处理器中的内部中断 Vector/Priority 寄存器相对应,而 mpic->isus 数组的 12~14 项与 MPC8541 处理器的外部中断 Vector/Priority 寄存器相对应。因此这段程序执行完毕后,MPIC 中断控制器硬件中断号 0~31 将与 MPC8541 处理器的内部中断号 0~31 一一对应;硬件中断号 48~59 与 MPC8541 处理器的外部中断号 0~11 一一对应。

举例说明,如果 MPC8541 处理器的 TSEC1 transmit interrupt 中断来临时,将引发中断响应周期,此时中断服务程序将读取 IACK 寄存器获得硬件中断号,此时硬件中断号的值为 13;如果 MPC8541 处理器的引脚 IRQ7 有效时,将引发中断响应周期,此时中断服务程序将读取 IACK 寄存器获得硬件中断号,此时硬件中断号的值为 7+48=55。

```
/* mpic_init 函数源代码片段 3 */
  /* Init spurrious vector */
  mpic_write(mpic->gregs,MPIC_INFO(GREG_SPURIOUS),MPIC_VEC_SPURRIOUS);
```

```
    /* Disable 8259 passthrough, if supported */
    if (! (mpic->flags & MPIC_NO_PTHROU_DIS))
      mpic_write(mpic->gregs, MPIC_INFO(GREG_GLOBAL_CONF_0),
           mpic_read(mpic->gregs, MPIC_INFO(GREG_GLOBAL_CONF_0))
           | MPIC_GREG_GCONF_8259_PTHROU_DIS);

    /* Set current processor priority to 0 */
    mpic_cpu_write(MPIC_INFO(CPU_CURRENT_TASK_PRI), 0);
} /* End mpic_init */
```

mpic_init 函数最后将初始化 Spurious 中断向量，关闭中断控制器 PIC 的 Passthrough 功能，将 CTPR 寄存器设置为 0，使能所有外部中断。

mpic_init 函数执行完毕后，mpc85xx_cds_pic_init 函数随之执行完毕，从而完成整个中断控制器的初始化。

6.3 设备驱动程序与外部中断处理系统的挂接

在 Linux 系统中，许多外部设备需要使用中断资源。为此，Linux 系统提供了一套机制，可以将设备驱动程序的中断服务例程与 Linux PowerPC 的外部中断处理程序挂接，即与 ExternalInput 函数挂接在一起。当外部中断发生时，ExternelInput 函数可以调用设备驱动程序的中断服务例程处理相应的中断源。

在 Linux PowerPC 中，程序员可以使用 request_irq 函数将设备驱动程序与外部中断处理函数 ExternalInput 挂接在一起。Linux PowerPC 为了实现这种挂接功能，设置了一系列全局变量和函数。

6.3.1 外部中断描述符表 irq_desc

在 Linux 中断系统中，全局数组 irq_desc 的地位举足轻重。该数组保存了 Linux 系统中所有外部中断描述符的信息。Linux 系统将该数组简称为外部中断描述符表。Linux 系统的外部中断描述符表 irq_desc 在 ./kernel/irq/handle.c 文件中定义，其源代码如下：

```
/*
 * Linux has a controller-independent interrupt architecture.
 * Every controller has a 'controller-template', that is used
 * by the main code to do the right thing. Each driver-visible
 * interrupt source is transparently wired to the appropriate
 * controller. Thus drivers need not be aware of the
 * interrupt-controller.
 *
 * The code is designed to be easily extended with new/different
 * interrupt controllers, without having to do assembly magic or
 * having to touch the generic code.
```

```
 *
 * Controller mappings for all interrupt sources:
 */
struct irq_desc irq_desc[NR_IRQS] __cacheline_aligned = {
  [0... NR_IRQS-1] = {
    .status = IRQ_DISABLED,
    .chip = &no_irq_chip,
    .handle_irq = handle_bad_irq,
    .depth = 1,
    .lock = __SPIN_LOCK_UNLOCKED(irq_desc->lock),
#ifdef CONFIG_SMP
    .affinity = CPU_MASK_ALL
#endif
  }
};
```

由此可以发现,外部中断描述符表一共有 NR_IRQS 个 Entry,并使用 irq_desc 结构描述每一个 Entry。其中,NR_IRQS 为当前处理器系统软件中断号的最大值。外部中断描述符表的每一个 Entry 与 Linux 系统中的软件中断一一对应,该表描述 Linux 系统的每一个软件中断号。在 Linux PowerPC 中,外部中断描述符表 irq_desc 是联系外部中断处理函数和外部设备中断服务例程之间的纽带。

- 设备驱动程序使用 request_irq 函数进行中断申请时,需要将中断服务例程的入口地址放入外部中断描述符表中。
- 外部中断处理函数 ExternelInput 开始执行时,需要从外部中断描述符表中获得相应的中断服务例程。

Linux 系统使用 irq_desc 结构表示外部中断描述符表 irq_desc 的每一个 Entry,irq_desc 结构的定义在 ./include/linux/irq.h 文件中,其代码如下:

```
struct irq_desc {
  irq_flow_handler_t   handle_irq;
  struct irq_chip      *chip;
  void          *handler_data;
  void          *chip_data;
  struct irqaction   *action;        /* IRQ action list */
  unsigned int       status;         /* IRQ status */

  unsigned int       depth;          /* nested irq disables */
  unsigned int       wake_depth;     /* nested wake enables */
  unsigned int       irq_count;      /* For detecting broken IRQs */
  unsigned int       irqs_unhandled;
  spinlock_t   lock;
#ifdef CONFIG_SMP
```

```
    cpumask_t       affinity;
    unsigned int    cpu;
#endif
#if defined(CONFIG_GENERIC_PENDING_IRQ) || defined(CONFIG_IRQBALANCE)
    cpumask_t       pending_mask;
#endif
#ifdef CONFIG_PROC_FS
    struct proc_dir_entry *dir;
#endif
    const char       *name;
} ____cacheline_aligned;
```

结构 irq_desc 的参数如下：

(1) handle_irq 参数指向外部中断处理函数。在 Linux PowerPC 中，handle_irq 参数指向的函数包括 handle_simple_irq、handle_level_irq、handle_fasteoi_irq 和 handle_edge_irq 等。

(2) chip 参数是 irq_chip 结构的参数。此参数指向一组对中断控制器 PIC 操作的函数集，如 ack，mask，mask_ack，unmask 和 eoi 函数等。

(3) handler_data 是 void 类型的指针。Linux PowerPC 使用该参数存放中断控制器描述符信息。对于 MPIC 中断控制器，handler_data 保存 mpic 结构或者其他中断控制器的指针。具有两个以上中断控制器的 Linux 系统，需要对 handler_data 参数赋值。

(4) chip_data 也是 void 类型的指针。该指针指向 irq_desc 结构的一些私有数据，比如在 MPC83XX 处理器中，该指针指向 QE 使用的 qe_ic 结构。

(5) action 参数指向 irqaction 结构类型的指针。该参数描述中断处理函数，设备驱动程序的中断服务例程挂接到这个参数。

(6) status 参数描述中断的状态。在 Linux PowerPC 中，中断可以有以下几种状态：

- IRQ_INPROGRESS，表示该中断正在处理中。
- IRQ_DISABLED，表示该中断被禁止。
- IRQ_PENDING，表示该中断正在等待处理。
- IRQ_REPLAY 表示中断被重发，但是并没有得到响应。
- IRQ_AUTODETECT 表示自动获取中断号。在驱动程序中，可以使用 probe_irq_on 函数自动获得软件中断号。当进行中断号自动获取时，系统会将所有未分配的硬件中断打开，再使相应的设备产生一个中断，然后通过扫描 irq_desc 中的所有中断源，判断有没有发生过中断，从而获得该设备的软件中断号。对此段代码有兴趣的读者可以参考 ./kenel/irq/autoprobe.c 文件中的 probe_irq_on 函数。
- IRQ_WAITING 表示自动获取中断号的过程尚未完毕。

6.3.2 软硬件中断号的映射

在设备驱动程序中，用户可以使用 request_irq 函数将该外部设备的中断服务例程挂接到外部中断处理程序中。外部中断处理程序，即 ExternelInput 函数，可以直接处理硬件中断号，而 request_irq 函数使用软件中断与外部中断程序挂接。因此 Linux PowerPC 必须采取某种

机制建立软硬件中断的映射。下文以 MPC8541 处理器的 TSEC 设备驱动程序为例，说明 Linux PowerPC 如何建立这一映射关系。

TSEC 驱动程序调用 ucc_geth_probe->irq_of_parse_and_map 函数，完成软件中断与硬件中断间的映射关系，ucc_geth_probe 函数的源代码在 ./driver/net/ucc_geth.c 文件中。

TSEC 设备驱动程序为实现软硬件之间的映射关系煞费苦心。irq_of_parse_and_map 函数最初在 gfar_of_init 函数中调用，建立这种映射关系。后来又将 irq_of_parse_and_map 函数改在 ucc_geth_probe 函数中调用，将 irq_of_parse_and_map 函数放在此处使用也不一定完全合适，因为不能要求一个普通的设备驱动程序员对 Linux PowerPC 的中断系统有这么深入的认识才能编写设备驱动程序。比较好的办法是让设备驱动程序员仅需要了解软件中断号的含义，就能够编写相应的设备驱动程序的中断服务例程。因此这种在 TSEC 设备驱动程序中实现软硬件中断号映射的方法可能很快就会被淘汰。

以上这段文字主要的目的是提醒读者注意，设备驱动程序中使用 request_irq 函数申请中断资源之前，Linux PowerPC 需要将 request_irq 函数使用的软件号与硬件号之间建立映射关系。Linux PowerPC 使用 irq_of_parse_and_map 函数建立这种软硬件之间的映射关系，该函数在 ./arch/powerpc/kernel/irq.c 文件中，其详细描述如下：

```
unsigned int irq_of_parse_and_map(struct device_node *dev,int index)
{
    struct of_irq oirq;

    if (of_irq_map_one(dev,index,&oirq))
        return NO_IRQ;

    return irq_create_of_mapping(oirq.controller,oirq.specifier,
                oirq.size);
}
EXPORT_SYMBOL_GPL(irq_of_parse_and_map);
```

下文以 TSEC 驱动程序为例说明该函数的使用。

irq_of_parse_and_map 函数调用 of_irq_map_one 函数从 TSEC 中断控制器的 dev_node 中获得 TSEC 使用的中断信息并填入 oriq 结构中。在 TSEC 中断控制器的 dev_node 中，与中断资源有关的信息在 ./arch/powerpc/bootdts/mpc8541cds.dts 文件中，如下所示：

```
ethernet@24000 {
    #address-cells = <1>;
    #size-cells = <0>;
    device_type = "network";
    model = "TSEC";
    compatible = "gianfar";
    reg = <24000 1000>;
    local-mac-address = [ 00 E0 0C 00 73 00 ];
    interrupts = <d 2 e 2 12 2>;
```

```
        interrupt-parent = <40000>;
        phy-handle = <2452000>;
    };
```

在该 dev_node 中,"interrupts = <d 2 e 2 12 2>"表示当前 TSEC 中断控制器一共有 3 个中断,其硬件中断号分别为 0x0d、0x0e 和 0x12,与 TSEC 发送中断、接收中断及"在发送/接收数据出现错误"中断相对应;"d"、"e"和"12"后的参数"2"表示中断属性。中断属性为 2 表示该 TSEC 中断源是高电平有效。

在获得 oriq 结构之后,irq_of_parse_and_map 函数调用 irq_create_of_mapping 函数完成软硬件中断号的映射。该函数的源代码如下:

```
unsigned int irq_create_of_mapping(struct device_node *controller,
            u32 *intspec, unsigned int intsize)
{
    struct irq_host *host;
    irq_hw_number_t hwirq;
    unsigned int type = IRQ_TYPE_NONE;
    unsigned int virq;

...

    /* If host has no translation, then we assume interrupt line */
    if (host->ops->xlate == NULL)
        hwirq = intspec[0];
    else {
        if (host->ops->xlate(host, controller, intspec, intsize,
                &hwirq, &type))
            return NO_IRQ;
    }
```

irq_creat_of_mapping 函数首先从 oriq.intspec[0]中或者调用 xlate 函数获得相应的硬件中断号 hwirq。基于 E500 内核的 Linux PowerPC,其 hwirq 的值为 intspec[0]。

```
    /* Create mapping */
    virq = irq_create_mapping(host, hwirq);
    if (virq == NO_IRQ)
        return virq;

    /* Set type if specified and different than the current one */
    if (type != IRQ_TYPE_NONE &&
        type != (get_irq_desc(virq)->status & IRQF_TRIGGER_MASK))
        set_irq_type(virq, type);
```

```
    return virq;
} /* irq_create_of_mapping */

EXPORT_SYMBOL_GPL(irq_create_of_mapping);
```

以上这段程序的主体是 irq _ create _ mapping 函数，该函数调用 irq _ find _ mapping 函数建立硬件中断号 hwirq 与软件中断号的之间的映射关系。如果 irq _ create _ mapping 函数执行失败，该函数的返回值为 NO _ IRQ，表示软硬件中断号映射失败；如果该函数执行成功将继续使用 set _ irq _ type 函数设置中断类型，最后将 irq _ create _ mapping 函数获得的软件中断号 virq 返回。

irq _ create _ mapping 函数有两个输入参数，一个是 host 参数，该参数描述 TSEC 控制器中断使用的 irq _ host 结构；另一个是 hwirq 参数，该参数存放 TSEC 控制器的硬件中断号，该硬件中断号从 dev _ node 中获得。irq _ create _ mapping 函数的详细解释如下：

```
unsigned int irq_create_mapping(struct irq_host *host,
        irq_hw_number_t hwirq)
{
    unsigned int virq, hint;
...
    if (host == NULL)
        host = irq_default_host;
    if (host == NULL) {
        printk(KERN_WARNING "irq_create_mapping called for"
             " NULL host, hwirq=%lx\n", hwirq);
        WARN_ON(1);
        return NO_IRQ;
    }
    /* Check if mapping already exist, if it does, call
     * host->ops->map() to update the flags
     */
    virq = irq_find_mapping(host, hwirq);
    if (virq != IRQ_NONE) {
        pr_debug("irq:-> existing mapping on virq %d\n", virq);
        return virq;
    }
```

irq _ create _ mapping 函数首先对 host 参数进行检查，如果 host 参数为 NULL，则使用全局变量 irq _ default _ host 作为缺省的 host 参数。如果 irq _ default _ host 变量仍然为空，则返回 NO _ IRQ。

随后，irq _ create _ mapping 函数调用 irq _ find _ mapping 函数，检查当前硬件中断号 hwirq 是否已经建立与软件中断号的映射关系。Linux PowerPC 定义了一个全局数组 irq _ map，这个数组也被称为软硬件中断映射表，该表与反向中断映射表相对应，在该数组中记载了所有软硬件中断号之间的映射关系。该表在初始化时建立，而反向中断映射表将在外部中

断处理程序中逐步建立。

irq_find_mapping 函数的主要功能是遍历 irq_map 数组，查找当前硬件号与软件中断号是否已经建立了映射关系，如果已经建立了映射关系，该函数将获得与此硬件中断号对应的软件中断号，之后该函数返回。如果 irq_find_mapping 函数没有获得软件中断号，即 virq 的值为 NO_IRQ 时，将继续执行以下程序。

这里提醒读者注意，在本段函数中，使用"virq != IRQ_NONE"作为软件中断号 virq 是否有效的条件有些不妥。虽然在 Linux PowerPC 中 IRQ_NONE 和 NO_IRQ 的值相等，都为 0，但是这里还是应该以"virq != NO_IRQ"作为判断条件，因为如果 irq_find_mapping 函数没有获得与当前硬件中断号对应的软件中断号，则其返回值为 NO_IRQ 而不是 IRQ_NONE。这有可能是编写这段程序员的一个笔误。irq_find_mapping 函数的源代码较易阅读，为节约篇幅，本书对此不做详细分析，但是建议读者务必阅读该函数的源代码。

```
/* Get a virtual interrupt number */
if (host->revmap_type == IRQ_HOST_MAP_LEGACY) {
  /* Handle legacy */
  virq = (unsigned int)hwirq;
  if (virq == 0 || virq >= NUM_ISA_INTERRUPTS)
    return NO_IRQ;
  return virq;
} else {
  /* Allocate a virtual interrupt number */
  hint = hwirq % irq_virq_count;
  virq = irq_alloc_virt(host,1,hint);
  if (virq == NO_IRQ) {
    pr_debug("irq:-> virq allocation failed\n");
    return NO_IRQ;
  }
}
pr_debug("irq:-> obtained virq %d\n",virq);
```

如果在 irq_find_mapping 函数中获得了软硬件之间的映射关系，这段程序将不会被执行。在这段程序中，首先判断当前 irq_host 使用的 revmap_type 参数的类型，即当前中断控制器是采用哪种方式映射硬软件中断号。对于基于 E500 内核的 Linux PowerPC，其 irq_host 的 revmap_type 参数为 IRQ_HOST_MAP_LINEAR，因此在 E500 内核中将执行 else 语句。

在 else 语句中这段程序调用 irq_alloc_virt 函数在 irq_map 数组中分配一个未使用的 Entry，最后在 irq_alloc_virt 函数返回时获得与该硬件中断号对应的软件中断号，如果在 irq_map 数组中没有找到合适的 Entry，则 irq_alloc_virt 函数返回值为 NO_IRQ。

```
/* Clear IRQ_NOREQUEST flag */
get_irq_desc(virq)->status &= ~IRQ_NOREQUEST;

/* map it */
smp_wmb();
```

```
    irq_map[virq].hwirq = hwirq;
    smp_mb();
    if (host->ops->map(host,virq,hwirq)) {
        pr_debug("irq:-> mapping failed,freeing \n");
        irq_free_virt(virq,1);
        return NO_IRQ;
    }
    return virq;
}
EXPORT_SYMBOL_GPL(irq_create_mapping);
```

在这段程序中,首先将外部中断描述符表 irq_desc 对应的 IRQ_NOREQUEST 位清除,表示该软件中断号可以被使用。之后将当前软件中断号与硬件中断号之前的映射关系存放到全局数组 irq_map 中。

最后调用 host->ops->map 函数进一步设置处理当前软件中断号的程序,对于基于 MPIC 控制器的 Linux PowerPC,该函数等效于 mpic_host_map 函数。mpic_host_map 函数是 Linux PowerPC 进行软硬件中断号映射重要的函数,其代码分析如下:

```
static int mpic_host_map(struct irq_host *h,unsigned int virq,
        irq_hw_number_t hw)
{
    struct mpic *mpic = h->host_data;
    struct irq_chip *chip;

    DBG("mpic:map virq %d,hwirq 0x%lx \n",virq,hw);
    if (hw == MPIC_VEC_SPURRIOUS)
        return -EINVAL;
    if (hw >= mpic->irq_count)
        return -EINVAL;

    /* Default chip */
    chip = &mpic->hc_irq;
```

mpic_host_map 函数共有 3 个参数:h,virq 和 hw。h 参数指向 irq_host 结构,该参数用来取得该函数使用的 mpic 数据结构的指针;virq 参数表示软件中断号,hw 参数表示硬件中断号。该函数正常返回时,返回值为 0 时。返回值为-EINVAL 时,表示输入参数错误。

在函数的开始部分,首先从 irq_host 结构中获得 mpic 的信息,然后检查输入参数 hw,最后从 mpic 结构中获得 hc_irq 参数,hc_irq 参数中保存了一些操作 MPIC 中断控制器内寄存器的一组函数。

```
    DBG("mpic:mapping to irq chip @%p \n",chip);

    set_irq_chip_data(virq,mpic);
```

```
    set_irq_chip_and_handler(virq,chip,handle_fasteoi_irq);

    /* Set default irq type */
    set_irq_type(virq,IRQ_TYPE_NONE);

    return 0;
} /* End mpic_host_map */
```

这段程序首先使用 set_irq_chip_data 函数将 mpic 结构保存在相应中断描述符 irq_desc 的 chip_data 参数中,以便以后使用,之后调用 set_irq_chip_and_handler 函数将相应中断描述符 irq_desc 的 handle 参数设置为 handle_fasteoi_irq 函数。

中断描述符 irq_desc 的 handle 参数设置完毕后,当外部中断处理程序处理这个中断源时,将首先执行 handle_fasteoi_irq 函数。set_irq_chip_and_handler 函数在 ./kernel/irq/chip.c 文件中,该函数的源代码如下:

```
void
set_irq_chip_and_handler(unsigned int irq,struct irq_chip *chip,
        irq_flow_handler_t handle)
{
  set_irq_chip(irq,chip);
  __set_irq_handler(irq,handle,0,NULL);
}
```

set_irq_chip 函数是设备驱动程序的中断服务例程与 MPIC 中断控制器进行挂接的要点。set_irq_chip 函数将终端描述符 irq_desc 中相应 Entry 的 chip 参数赋值为 mpic->hc_irq,即为 mpic_irq_chip。之后,在设备驱动中断服务例程可以使用 mpic_irq_chip 结构中提供的操作函数,访问 MPIC 中断控制器中的寄存器。__set_irq_handler 函数将中断描述符 irq_desc 相应 Entry 的 handle_irq 赋值为 mpic->hc_irq,即为 handle_fasteoi_irq。

Linux PowerPC 的中断子系统除了需要为 TSEC 控制器建立软硬件中断号映射之外,还需要对 TSEC 控制器的使用的软件中断号进行一些必要的初始化,之后设备驱动程序才能够使用 request_irq 函数将 TSEC 中断服务例程挂接到外部中断处理函数中。

Linux PowerPC 使用 platform bus 设备驱动统一管理 MPC8541 处理器中的 TSEC 控制器、FCC 控制器、DUART 等设备。

这个 platform bus 驱动程序的代码在 ./arch/powerpc/sysdev/fsl_soc.c 文件中。platform bus 设备驱动程序是 Linux 系统总线类驱动程序的一种,只是这类设备驱动程序没有实际的物理总线。有关 Linux 总线驱动程序的详细描述请参考 Jonathan,Alessandro 和 Greg 所著的 Linux Device Drivers 第 3 版。

Linux PowerPC 首先使用 gfar_of_init 函数,初始化 TSEC 控制器。在 gfar_of_init 函数中,首先由 OP API 类函数获得在 Open Firmware 中存放的 TSEC 控制器的描述信息,之后由 platform_device_register_simple 函数注册此 platform bus 驱动程序,然后使用 platform_device_add_data 函数将所有 TSEC 控制器的 platform_device 信息添加到 TSEC 控制器对应的 platform bus 设备驱动程序中,这个设备驱动程序由 do_initcalls 函数在 Linux 系统引导时

启动，有关 do _ initcalls 函数的详细介绍见 8.3.2 节。

gfar _ of _ init 函数执行完毕后，TSEC 控制器使用的 platform _ device 结构被设置完毕，此后 TSEC 设备驱动程序可以使用 platform _ get _ irq _ byname 函数从 platform bus 驱动程序中获得 TSEC 设备驱动程序使用的软件中断号。

6.3.3 request _ irq 函数

在 Linux 系统中，设备驱动程序使用 request _ irq 函数，将设备的中断服务例程与外部中断服务函数挂接，并使用 free _ irq 函数来释放这种挂接关系。request _ irq 和 free _ irq 函数的源代码在 ./kernel/irq/manage.c 文件。

1. request _ irq 函数的入口参数

```
int request_irq(unsigned int irq, irq_handler_t handler,
        unsigned long irqflags, const char * devname, void * dev_id)
{
```

request _ irq 函数共有 5 个参数，分别为 irq，handler，irqflags，devname 和 dev _ id。request _ irq 函数返回 0 时，表示外部设备的中断处理程序注册成功，返回-INVAL 表示当前软件中断号 irq 大于 NR _ IRQS(NR _ IRQS 为 Linux PowerPC 软件中断号的最大值)或 handler 等于空，返回-EBUSY 表示中断已经被占用且不能共享。

(1) handler 参数。handler 参数保存设备驱动程序使用的中断服务例程。当产生外部中断时，外部中断处理程序调用此函数来完成对此设备的中断处理。

中断服务例程一共有两个参数 irq 和 dev _ id。irq 参数存放软件中断号，dev _ id 参数为中断服务例程使用的设备 id 号。dev _ id 的主要作用是帮助中断处理程序获得指向该设备驱动程序的描述符指针，从而获得设备驱动程序需要的 priv 数据。

(2) irqflags 参数。该参数的值为 IRQF _ DISABLED 或者 IRQF _ SHARED。一些版本较老的设备驱动程序使用 SA _ SHIRQ 与 SA _ INTERRUPT。在 Linux 2.6 内核中，这两个值已经被 IRQF _ DISABLED 与 IRQF _ SHARED 替换。

当 irqflags 参数为 IRQF _ DISABLED 时，当前中断服务例程独占软件中断号，其他中断服务例程不能使用这个软件中断号，而且在此中断服务例程在运行时将屏蔽所有其他的外部中断。

当 irqflags 参数为 IRQF _ SHARED 时，当前中断服务例程可以与其他中断服务例程共享同一个软件中断号，而且在此中断服务例程运行时不屏蔽其他外部中断。在 Linux 系统中，如果某个软件中断号允许共享，那么所有使用这个软件中断号的设备驱动程序在调用 request _ irq 函数都需要将其 irqflags 参数设置为 IRQF _ SHARED。

(3) devname 和 dev _ id 参数表示设备名和设备 id 号。

(4) irq 参数用来存放设备驱动程序使用软件中断号。在设备驱动程序中，使用 request _ irq 函数时，需要使用软件中断号将中断服务例程挂接到外部中断处理函数 ExternalInput 中。在 Linux PowerPC 中，外部中断处理程序 ExternelInput 函数能够从中断控制器 PIC 直接获得的中断号为硬件中断号，但是在设备驱动程序中需要使用软件中断号，将中断服务例程与外部中断处理程序挂接在一起。

这是因为在Linux系统中，request _ irq函数不仅可以在Linux PowerPC中使用，而且可以在Linux ARM、Linux MIPS和Linux Pentium中使用。对于不同体系结构的处理器，硬件中断号的定义相差甚远，采用软硬件中断号分离的方法，有助于提高设备驱动程序的可移植性，也使得不同体系结构的处理器都可以使用request _ irq函数将中断服务例程挂接到相应的外部中断处理程序中。

Linux PowerPC支持Open Firmware结构。在MPC8541CDS处理器系统中，一些硬件中断号的描述被放入./arch/powerpc/boot/dts/mpc8541cds.dts文件中。如在mpc8541cds.dts文件中，I^2C总线使用的硬件中断号使用“interrupts = <1b 2>”，TSEC1控制器使用的硬件中断号使用“interrupts = <d 2 e 2 12 2>”，即I^2C总线使用的硬件中断号为0x1b = 27，而TSEC1控制器使用的硬件中断号为0xd = 13，0xe = 14和0x12 = 18。

MPC85XX处理器支持外部中断源。MPC85XX处理器提供了12个外部中断引脚，用户可以使用这些外部引脚，进行外部中断扩展。MPC85XX处理器的外部中断号的定义如下所示：

```
/* The 12 external interrupt lines */
#define MPC85xx_IRQ_EXT0    (48 + MPC85xx_OPENPIC_IRQ_OFFSET)
#define MPC85xx_IRQ_EXT1    (49 + MPC85xx_OPENPIC_IRQ_OFFSET)
#define MPC85xx_IRQ_EXT2    (50 + MPC85xx_OPENPIC_IRQ_OFFSET)
#define MPC85xx_IRQ_EXT3    (51 + MPC85xx_OPENPIC_IRQ_OFFSET)
#define MPC85xx_IRQ_EXT4    (52 + MPC85xx_OPENPIC_IRQ_OFFSET)
#define MPC85xx_IRQ_EXT5    (53 + MPC85xx_OPENPIC_IRQ_OFFSET)
#define MPC85xx_IRQ_EXT6    (54 + MPC85xx_OPENPIC_IRQ_OFFSET)
#define MPC85xx_IRQ_EXT7    (55 + MPC85xx_OPENPIC_IRQ_OFFSET)
#define MPC85xx_IRQ_EXT8    (56 + MPC85xx_OPENPIC_IRQ_OFFSET)
#define MPC85xx_IRQ_EXT9    (57 + MPC85xx_OPENPIC_IRQ_OFFSET)
#define MPC85xx_IRQ_EXT10   (58 + MPC85xx_OPENPIC_IRQ_OFFSET)
#define MPC85xx_IRQ_EXT11   (59 + MPC85xx_OPENPIC_IRQ_OFFSET
```

程序员在编写自定义设备的外部中断服务例程时，需要通过irq _ create _ mapping函数将这些硬件中断号转化为软件中断号后，才能使用request _ irq函数。

本书建议，需要使用自定义外设的程序员，将外设的描述信息放入dts文件中，然后在platform bus驱动程序中调用irq _ of _ parse _ and _ map函数，完成软硬件中断号的映射，而不是在设备驱动程序中直接调用irq _ create _ mapping函数，虽然采用这种方法最为简单。

2. request _ irq函数的代码分析

request _ irq函数的实现较为简单，其源代码在./kernel/irq/manage.c文件中：

```
int request_irq(unsigned int irq, irq_handler_t handler,
        unsigned long irqflags, const char *devname, void *dev_id)
{
    struct irqaction *action;
    int retval;
```

```
        if ((irqflags & IRQF_SHARED) && ! dev_id)
            return -EINVAL;
        if (irq >= NR_IRQS)
            return -EINVAL;
        if (irq_desc[irq].status & IRQ_NOREQUEST)
            return -EINVAL;
        if (! handler)
            return -EINVAL;
```

这段程序首先对进行 request_irq 函数的入口参数进行检查。

1）对 irqflags 参数进行检查。如果 IRQF_SHARED 位有效，则表示该外部设备与其他设备共享一个中断资源。在 Linux 系统中，一个软件中断号可能被多个中断服务例程共享。此时在 request_irq 函数中，必须使用设备驱动程序的 dev_id 参数，进一步区分是哪个设备正在使用这个软件中断号。因此当 irqflags 参数中的 IRQF_SHARED 位有效时，dev_id 不能为 NULL，否则 dev_id 可以为 NULL。

2）检查软件中断号，是否大于其可能的最大值 NR_IRQS。检查在全局变量 irq_desc 中与此中断号对应 Entry 的状态是否有效。检查中断服务例程是否为空。

```
        action = kmalloc(sizeof(struct irqaction),GFP_ATOMIC);
        if (! action)
            return -ENOMEM;

        action->handler = handler;
        action->flags = irqflags;
        cpus_clear(action->mask);
        action->name = devname;
        action->next = NULL;
        action->dev_id = dev_id;

        select_smp_affinity(irq);

        retval = setup_irq(irq,action);
        if (retval)
            kfree(action);

        return retval;
    } /* End request_irq */
```

这段源代码的执行流程如下：

1）首先为 action 变量分配物理空间，action 为 irqaction 类型的变量。

2）初始化 action 变量，将中断服务例程的 flags 和 handle 参数加入到 action 变量中。

3）设置该中断服务例程的亲和性属性，即由处理器的哪个 CPU 运行该中断服务例程。

4）调用 setup_irq 函数将 action 变量中包含的信息加入到 irq_desc 数组中。

3. setup _ irq 函数

setup _ irq 函数将中断服务例程加入到中断描述符表 irq _ desc 中，完成设备驱动程序与外部中断处理系统的挂接。

setup _ irq 函数一共有两个输入参数。

(1) irq 参数。该参数用来存放设备驱动程序使用的软件中断号。

(2) new 用来存放与 irq 参数对应的 irqaction 结构，该结构在 request _ irq 函数中赋值。

setup _ irq 函数将存放在参数 new 中的 irqaction 结构合理地加入到 irq _ desc 外部中断描述符表中，其源代码如下：

```
int setup_irq(unsigned int irq,struct irqaction *new)
{
  struct irq_desc *desc = irq_desc + irq;
  struct irqaction *old, **p;
  const char *old_name = NULL;
  unsigned long flags;
  int shared = 0;

  if (irq >= NR_IRQS)
    return -EINVAL;

  if (desc->chip == &no_irq_chip)
    return -ENOSYS;

  if (new->flags & IRQF_SAMPLE_RANDOM) {
    rand_initialize_irq(irq);
  }
  spin_lock_irqsave(&desc->lock,flags);
  p = &desc->action;
  old = *p;
  if (old) {
    if (!((old->flags & new->flags) & IRQF_SHARED) ||
      ((old->flags ^ new->flags) & IRQF_TRIGGER_MASK)) {
      old_name = old->name;
      goto mismatch;
    }

    do {
      p = &old->next;
      old = *p;
    } while (old);
    shared = 1;
  }
```

这段代码的执行说明如下。

1）首先根据软件中断号 irq 查找外部中断描述符表 irq_desc，得到相应的中断描述符 desc。

2）接着进行一些必要的参数检查。

3）对临界数据 desc 加锁，并禁止中断。

4）对 desc->action 进行检查。如果 desc->action 的值为 NULL，表示没有其他设备驱动程序使用过 request_irq 函数对此软件中断号进行注册。如果其值不为空则表示曾经有设备驱动程序注册过此中断号。

5）之后，这段程序进一步检查当前需要注册的中断处理程序和 desc->action 中的中断处理程序是否可以共享，并且这两个中断处理程序是否采用了相同的触发方式。如果这两个条件都为真，则将新的中断处理程序加入到老的中断处理程序之后，并将 shared 变量赋值为 1，表示当前至少有两个设备驱动程序共享一个软件中断号。如果此时新老中断处理程序出现了共享错误，将会调转到 mismatch 标签处执行。

```
    *p = new;
    if (! shared) {
        irq_chip_set_defaults(desc->chip);

        /* Setup the type (level, edge polarity) if configured: */
        if (new->flags & IRQF_TRIGGER_MASK) {
            if (desc->chip && desc->chip->set_type)
                desc->chip->set_type(irq,
                        new->flags & IRQF_TRIGGER_MASK);
            else
                printk(KERN_WARNING "No IRQF_TRIGGER set_type "
                        "function for IRQ %d (%s)\n", irq,
                        desc->chip ? desc->chip->name :
                        "unknown");
        } else
            compat_irq_chip_set_default_handler(desc);

        desc->status &= ~(IRQ_AUTODETECT | IRQ_WAITING |
                IRQ_INPROGRESS);

        if (! (desc->status & IRQ_NOAUTOEN)) {
            desc->depth = 0;
            desc->status &= ~IRQ_DISABLED;
            if (desc->chip->startup)
                desc->chip->startup(irq);
            else
                desc->chip->enable(irq);
        } else
            /* Undo nested disables: */
```

```
        desc->depth = 1;
    } /* End if (! shared) */
```

- 如果shared变量不为1,则使用irq_chip_set_defaults函数对desc->chip中的操作函数进行初始化。irq_chip_set_defaults函数在./kernel/irq/chip.c文件中,该函数将desc->chip中的enable、disable、startup、shutdown操作函数、name参数和end参数赋值。在Linux PowerPC中,desc->chip的enable和disable函数为mpic_unmask_irq和mpic_mask_irq函数。
- 随后根据flags参数设置与当前软件中断号对应中断源的触发条件。
- 将desc->status的IRQ_AUTODETECT、IRQ_WAITING、IRQ_INPROGRESS位清除。
- 如果desc->status参数允许使能中断源,即IRQ_NOAUTOEN位为0,则调用startup函数或者调用enable函数使能外部中断。MPIC中断处理程序的startup函数和enable函数将调用mpic_unmask_irq函数将IIVPR0~31或者EIVPR0~11中的某个寄存器的MSK位清零,使能相应中断源。

```
    /* Reset broken irq detection when installing new handler */
    desc->irq_count = 0;
    desc->irqs_unhandled = 0;
    spin_unlock_irqrestore(&desc->lock,flags);

    new->irq = irq;
    register_irq_proc(irq);
    new->dir = NULL;
    register_handler_proc(irq,new);

    return 0;

  mismatch:
    if (! (new->flags & IRQF_PROBE_SHARED)) {
      printk(KERN_ERR "IRQ handler type mismatch for IRQ %d\n",irq);
      if (old_name)
        printk(KERN_ERR "current handler: %s\n",old_name);
      dump_stack();
    }
    spin_unlock_irqrestore(&desc->lock,flags);
    return -EBUSY;
  } /* End setup_irq */
```

- 最后,setup_irq函数打开自旋锁desc->lock。
- 将中断信息注册到proc文件系统中,此后Linux系统的使用者可以使用"cat /proc/interrupts"命令显示Linux内核中所有中断信息。

如果setup_irq函数执行失败,将跳转到mismatch标签处,并返回-EBUSY。如果此时在

执行 setup _ irq 函数时，出现了共享错误，则调用 dump _ stack 函数，将当前进程使用的核心堆栈打印出来。

6.4 Linux PowerPC 对外部中断的处理

对于 PowerPC E500 内核，Linux PowerPC 对外部中断的处理共分为两个步骤。

(1) 将 E500 内核中的 IVORs 寄存器与 IVPR 寄存器赋值，使得对应的中断或者异常由指定的函数进行处理。

(2) 当外部中断事件来临时，由指定的外部中断处理函数处理相应的中断事件。

对于 E500 内核，Linux PowerPC 在引导初期将中断向量初始化。Linux PowerPC 使用宏定义 SET _ IVOR 初始化 PowerPC E500 内核中的 IVORx 寄存器。该宏定义的源代码在 ./arch/powerpc/kernel/head _ booke.h 文件中。在本章的起始部分讲述了如何使用宏定义 SET _ IVOR 初始化 IVORx 寄存器和 IVPR 寄存器，本节主要介绍 Linux PowerPC 的外部中断处理函数，即 ExternalInput 函数。

发生外部中断时，Linux PowerPC 将调用 ExternalInput 函数处理外部中断。为此 Linux PowerPC 需要使用宏 EXCEPTION 初始化外部中断处理函数 ExternalIuput。

```
EXCEPTION(0x0500,ExternalInput,do _ IRQ,EXC _ XFER _ LITE)
```

宏 EXCEPTION 的源代码在 ./arch/powerpc/kernel/head _ booke.h 文件中，将该宏展开可以得到以下程序：

```
    .align 5;
ExternalInput:;
    NORMAL _ EXCEPTION _ PROLOG;
    addi r3,r1,STACK _ FRAME _ OVERHEAD;
    EXC _ XFER _ LITE (0x0500,do _ IRQ);
```

- Linux PowerPC 要求 ExternalInput 函数的程序地址需要至少 4 字节对界，这是因为 IVOR 寄存器的低 4 位恒为 0。
- 之后这段程序调用宏 NORMAL _ EXCEPTION _ PROLOG 为进入中断程序做必要的准备工作。
- 更改通用寄存器 GPR3 的值，最后调用宏 EXC _ XFER _ LITE 进行中断处理。此时寄存器 GPR3 的值将指向中断堆栈中的 pt _ regs 参数，该参数将作为 do _ IRQ 函数的输入参数。

6.4.1 宏 NORMAL _ EXCEPTION _ PROLOG

在 Linux PowerPC 中，大多数异常处理程序使用宏 NORMAL _ EXCEPTION _ PROLOG 进行初始化。该宏的源代码在 ./arch/powerpc/kernel/head _ booke.h 文件中。

宏 NORMAL _ EXCEPTION _ PROLOG 主要的作用有以下两个：

(1) 确定外部中断处理程序使用的堆栈空间。

(2) 将中断处理程序中使用的通用寄存器和状态寄存器压入中断堆栈，为 do _ IRQ 函数

提供运行空间。

1. **确定中断服务程序使用的堆栈空间**

对于基于 E500 内核的 PowerPC 处理器，Linux PowerPC 并没有使用独立的中断堆栈结构，而是使用进程的核心堆栈作为中断堆栈。Linux PowerPC 确定中断处理程序堆栈的代码如下：

```
#define NORMAL_EXCEPTION_PROLOG                \
    mtspr SPRN_SPRG0,r10;                      \
    mtspr SPRN_SPRG1,r11;                      \
    mtspr SPRN_SPRG4W,r1;                      \
    mfcr r10;                                  \
```

这段程序首先将寄存器 r10、r11 和 r1 保存在 SPRG0，SPRG1 和 SPRG4W 中。在下面的程序中，需要使用 r10，r11 作为缓冲，以确定中断程序使用的堆栈。在确定中断处理程序使用的堆栈之前，只能使用 PowerPC 处理器提供的 SPR 寄存器保存通用寄存器 GPR。在 Linux 系统中，SPR 寄存器被用来保存中断或者异常处理程序的通用寄存器，其他的内核程序不得使用这些寄存器。之后这段程序使用 r10 保存 E500 内核的 CR 寄存器。

```
    mfspr r11,SPRN_SRR1;                       \
    andi. r11,r11,MSR_PR;                      \
    beq 1f;                                    \
    mfspr r1,SPRN_SPRG3;                       \
    lwz r1,THREAD_INFO-THREAD(r1);             \
    addi r1,r1,THREAD_SIZE;                    \
1:subi r1,r1,INT_FRAME_SIZE;                   \
```

这段程序从 SRR1 寄存器中获得进入中断处理程序之前的 MSR 寄存器，注意在进入中断处理程序之后，MSR 寄存器的有些位被自动清零。因此如果程序员需要操作“进入中断处理程序之前”的 MSR 寄存器时，需要将 MSR 寄存器从 SRR1 寄存器中恢复。

之后这段程序对“进入中断处理程序之前”的 MSR 寄存器的 PR 位进行判断。如果 PR 位为 1 表示进程是在 Linux 用户空间中运行被中断的，如果 PR 位 0 表示进程是在 Linux 核心空间内运行被中断的。

根据 PR 位的不同，中断处理程序需要对 r1 寄存器作不同的处理。E500 内核使用寄存器 r1 作为堆栈指针寄存器。在 Linux PowerPC 中，中断处理程序不能直接使用用户空间的堆栈指针寄存器，因为在 Linux 系统中，用户空间的虚拟内存不一定在实际的物理内存中。因此在 Linux PowerPC 中，外部中断处理程序需要使用用户进程的核心栈段作为中断处理程序的栈段。

当 PR 位为 1 时，中断处理程序将堆栈指针寄存器 r1 指向用户进程的核心堆栈的顶部，因为此时用户没有在 Linux 系统的核心态中运行，在核心堆栈中并没有保存任何有效数据。Linux PowerPC 的中断处理程序采用以下算法获得用户进程的核心栈段的顶部：

```
r1 = mem(THREAD_INFO-THREAD + SPRG3) + THREAD_SIZE
```

其中寄存器 SPRG3 存放的是被中断进程的 thread 参数的地址，因此 SPRG3-THREAD 所得的值为进程描述符 task_struct 的地址。此地址在加上 THREAD_INFO 可以得到当前进

程描述符 thread _ info 参数的地址。最后使用这个地址中存放的数据加上 THREAD _ SIZE 就可以得到当前进程核心栈段的栈顶地址。在 Linux PowerPC 中,thread _ info 参数与核心堆栈共享一个 8KB 大小的空间,如图 5-3 所示。

对于 E500 内核,这段程序可以直接从 r2 寄存器中,直接获得进程描述符的地址而不需要使用"SPRG3-THREAD"。但是对于对于其他体系的 PowerPC 处理器,如基于 603E 内核,只能使用 SPRG3 首先获得 thread 参数的物理地址,之后再找到进程描述符的物理地址,详情见 5.1.1 节。

当 PR 位为 0 时,中断处理程序可以直接使用 r1 的值作为中断服务程序的堆栈指针,即直接使用当前进程的核心堆栈空间。

最后,这段程序使用"subi r1,r1,INT _ FRAME _ SIZE"指令为中断处理程序开辟堆栈空间,其大小为 INT _ FRAME _ SIZE = 16 + sizeof(struct pt _ regs)。

Linux 2.6.20 中支持独立的中断堆栈,即中断堆栈可以不使用进程的核心堆栈。这样做的好处是中断处理程序溢出时不对核心堆栈产生影响。这一技术在许多嵌入式操作系统中已经广泛采用。但在 Linux 系统中,并非所有的体系结构都支持中断堆栈,Linux PowerPC 支持独立的中断堆栈,但是 Linux PowerPC 并没有在 2.6.20 版本中支持 32 位 PowerPC 处理器提供中断堆栈。对中断堆栈有兴趣的读者可以阅读 ./arch/powerpc/kernel/setup _ 64.c 中的 irqstack _ early _ init 函数了解有关中断核心堆栈的初始化,本书对此不进行讨论。

2. 保存通用寄存器和状态寄存器

```
mr      r11,r1;                         \
stw   r10,_CCR(r11);                    \
stw   r12,GPR12(r11);                   \
stw   r9,GPR9(r11);                     \
mfspr   r10,SPRN_SPRG0;                 \
stw   r10,GPR10(r11);                   \
mfspr   r12,SPRN_SPRG1;                 \
stw   r12,GPR11(r11);                   \
mflr r10;                               \
stw   r10,_LINK(r11);                   \
```

以上这段程序的执行流程如下:

- 将寄存器 r1 的值赋予寄存器 r11,此时寄存器 r11 将指向当前堆栈指针。
- 将寄存器 r10 寄存器保留的 CR 寄存器存放入堆栈的相应位置。
- 将寄存器 r12,r9 存放入堆栈的相应位置。
- 将保存在 SPRG0 中的寄存器 r10 存放入堆栈的相应位置。
- 将保存在 SPRG1 中的寄存器 r11 存放入堆栈的相应位置。
- 将寄存器 LR 存放入堆栈的相应位置。

```
mfspr r10,SPRN_SPRG4R;                  \
mfspr   r12,SPRN_SRR0;                  \
stw   r10,GPR1(r11);                    \
```

```
mfspr  r9,SPRN_SRR1;                  \
stw  r10,0(r11);                      \
rlwinm r9,r9,0,14,12;                 \
stw  r0,GPR0(r11);                    \
SAVE_4GPRS(3,r11);                    \
GPRS(7,r11)
```

- 将保存在 SPRG4W 的寄存器 r1 放入堆栈的相应位置。
- 将 SRR0 中存放的中断返回地址存入寄存器 r12，将 SRR1 中存放的 MSR 的值存入寄存器 r9。
- 将进入中断之前的寄存器 r1(堆栈指针)放入当前中断栈帧的首部。
- 清除 r9 寄存器中的第 13 位，即清除对应 MSR 寄存器中的 WE 位。
- 将寄存器 r0 存放入堆栈的相应位置。
- 将寄存器 r3～r8 存放入堆栈的当前位置。

由以上代码分析可以发现，宏 NORMAL_EXCEPTION_PROLOG 执行完毕后，Linux 系统将为中断处理程序建立堆栈，并保存好 r3～r12 寄存器，最后将进入中断处理程序之前的 MSR 寄存器和中断返回地址分别存放入寄存器 r9 和 r12 中。

6.4.2 宏 EXC_XFER_LITE

宏 EXC_XFER_LITE 的定义在 ./arch/powerpc/kernel/head_booke.h 文件中，其源代码详解如下。该宏的主要作用是调用 do_IRQ 函数处理外部中断。

```
#define EXC_XFER_LITE(n,hdlr)        \
  EXC_XFER_TEMPLATE(hdlr,n+1,MSR_KERNEL,NOCOPY,transfer_to_handler, \
    ret_from_except)

#define EXC_XFER_TEMPLATE(hdlr,trap,msr,copyee,tfer,ret)  \
  li  r10,trap;                                   \
  stw  r10,_TRAP(r11);                            \
  lis  r10,msr@h;                                 \
  ori  r10,r10,msr@l;                             \
  copyee(r10,r9);                                 \
  bl tfer;                                        \
  .long hdlr;                                     \
  .long ret
```

宏 EXC_XFER_LITE 将调用宏 EXC_XFER_TEMPLATE。通过对以上源代码分析，读者可以发现 EXC_XFER_LITE (0x0500,do_IRQ)等效于以下语句：

```
li r10,0x501;                      \
stw r10,TRAP(r11);                 \
lis r10,MSR_KERNEL @h;             \
ori r10,r10,MSR_KERNEL @l;         \
```

```
bl transfer_to_handler;                    \
.long do_IRQ;                              \
.long ret_from_except
```

- 这段程序将 0x501 存放到当前堆栈的 trap 寄存器中。
- 将 MSR_KERNEL 的值存入寄存器 r10。MSR_KERNEL 的值请参考 ./include/asm-powerpc/reg_booke.h 文件。
- 调用 transfer_to_handler 函数,do_IRQ 函数及 ret_from_except。其中 transfer_to_handler 函数在 ./arch/powerpc/kernel/entry_32.S 文件中,该函数的源代码如下所示:

```
transfer_to_handler:
  stw   r2,GPR2(r11)                      /*  Line 1   */
  stw   r12,_NIP(r11)                     /*  Line 2   */
  stw   r9,_MSR(r11)                      /*  Line 3   */
  andi. r2,r9,MSR_PR                      /*  Line 4   */
  mfctr r12                               /*  Line 5   */
  mfspr r2,SPRN_XER                       /*  Line 6   */
  stw   r12,_CTR(r11)                     /*  Line 7   */
  stw   r2,_XER(r11)                      /*  Line 8   */
  mfspr r12,SPRN_SPRG3                    /*  Line 9   */
  addi  r2,r12,-THREAD                    /*  Line 10  */
  tovirt(r2,r2)                           /*  Line 11  */
  beq   2f                                /*  Line 12  */
  addi  r11,r1,STACK_FRAME_OVERHEAD       /*  Line 13  */
  stw   r11,PT_REGS(r12)                  /*  Line 14  */
  lwz   r12,THREAD_DBCR0(r12)             /*  Line 15  */
  andis.  r12,r12,DBCR0_IC@h              /*  Line 16  */
  beq+  3f                                /*  Line 17  */
```

- Line 1:将寄存器 r2 存入堆栈的相应位置,寄存器 r2 保存着 current 指针。
- Line 2:将保存着中断返回地址的寄存器 r12 压入堆栈,transfer_to_handler 函数结束后将调用 do_IRQ 函数而不是结束中断处理程序,因此需要将中断返回地址首先压入堆栈。
- Line 3:将保存在 r9 寄存器中,"进入中断处理程序之前"的 MSR 寄存器压入堆栈。
- Line 4:判断中断前的程序运行在超级用户还是普通用户模式,之后决定 Line12 的 beq 2f 语句是否转移,"2f"标签与这条语句距离很远。
- Line 5:将 CTR 寄存器的值存入 r12 寄存器。
- Line 6:将 XER 寄存器的值存入 r2 寄存器。
- Line 7:将 CTR 寄存器的值存入堆栈的相应位置。
- Line 8:将 XER 寄存器的值压栈。
- Line 9:从 SPRG3 寄存器中获得被中断进程描述符 thread 参数的地址,然后将该值存入寄存器 r12 中。

- Line 10：获得被中断进程的描述符指针，然后将该值存放入寄存器 r2 中。
- Line 11：对于 E500 内核，函数 tovirt(r2,r2)相当于 r2 = r2。这段代码通过函数 tovirt 终于获得了 current 指针，在 E500 内核中实际上不必如此。但是对于基于 603E 的 PowerPC 处理器，在进入中断服务程序时，MMU 将被关闭，因此必须通过 SPGR3 寄存器中保存的 task _ struct ->thread 参数的物理地址，找到 task _ struct 结构的物理地址，之后再通过 tovirt 函数将这个物理地址转换为虚拟地址。
- Line 12：如果被中断进程是在内核空间运行的则跳转到 2f，否则继续。
- Line 13：将当前堆栈中存放的 E500 内核的通用寄存器组指针存放到 r11 寄存器中。如上述代码所示，中断堆栈的大小为 STACK _ FRAME _ OVERHEAD + sizeof(struct pt _ regs)，因此将 r1 加上常数 STACK _ FRAME _ OVERHEAD，可以得到堆栈中存放的 E500 内核的常用寄存器组指针。
- Line 14：将 r11 寄存器的值更新进程描述符的 thread ->reg 参数。
- Line 15～16：如果被中断进程（当然是用户进程）没有被 ptrace 跟踪，则跳转到 3f，否则将执行以下程序清除所有 Debug 事件。

```
li r12,-1       /* clear all pending debug events */
mtspr SPRN _ DBSR,r12
lis r11,global _ dbcr0@ha
tophys(r11,r11)
addi r11,r11,global _ dbcr0@l
lwz r12,0(r11)
mtspr SPRN _ DBCR0,r12
lwz r12,4(r11)
addi r12,r12,-1
stw r12,4(r11)
b 3f
```

这段程序清除所有的 Debug 事件后，跳转到 3f。

```
2:
  lwz r9,THREAD _ INFO-THREAD(r12)
  cmplw r1,r9          /* if r1 <= current->thread _ info */
  ble- stack _ ovf     /* then the kernel stack overflowed */
```

如果被中断进程在 Linux 内核中运行，将执行这段代码，这段代码用来判断寄存器 r1 是否小于 current->thread _ info，如果小于则表示核心堆栈溢出。由上文得知，在 Linux PowerPC 中，核心堆栈与进程描述符的 thread _ info 共享一个 8 KB 大小的空间。当核心堆栈与 thread _ info 占用的空间重合时，导致核心堆栈溢出。

```
  .globl transfer _ to _ handler _ cont
transfer _ to _ handler _ cont:
3:
  mflr r9
  lwz   r11,0(r9)    /* virtual address of handler */
```

```
lwz  r9,4(r9)      /* where to go when done */
mtspr SPRN_SRR0,r11
mtspr SPRN_SRR1,r10
mtlr  r9
SYNC
RFI                /* jump to handler,enable MMU */
```

上述代码是 transfer_to_handler 函数最为核心的部分。本段程序合理地使用 RFI 指令，巧妙地调用了 do_IRQ 函数与 ret_from_except 函数。

- 首先将 LR 寄存器保存到 r9 寄存器。此 LR 寄存器的值为宏 EXC_XFER_LITE (0x0500,do_IRQ)中,bl transfer_to_handler 的下一条语句的地址。
- 将寄存器 r9 地址内的数据赋予 r11,即 .long do_IRQ。
- 将寄存器 r9+4 地址内的数据赋予 r9,即 .long ret_from_except。
- 将寄存器 r11 的值赋予 SRR0 寄存器,因此执行 RFI 指令将跳转到 do_IRQ 函数。
- 将寄存器 r10 的值赋予 SRR1 寄存器。
- 将保存在 r9 寄存器中的数据赋予 LR 寄存器,此时 LR 寄存器将保存 ret_from_except 函数的地址。
- 使用 RFI 指令将程序跳转到 do_IRQ 函数。此时 LR 寄存器中的为 ret_from_except,因此当 do_IRQ 结束后将继续调用 ret_from_except 函数。

对于习惯了单进程环境下编程的程序员,RFI 指令很容易被误解。这些程序员经常认为 RFI 指令结束后,中断将处理完毕,处理器会返回到被中断的程序中继续执行。

许多在 Linux 这个标准多进程操作系统环境下编程的程序员,实际上仍然沿袭着单进程的编程习惯。实际上,RFI 指令和中断处理程序结束没有什么必然联系。在 Linux 系统中,中断结束后也不一定要回到被中断的进程中运行。在 ret_from_except 函数执行完毕后将调用 schedule 函数。schedule 函数将决定在中断处理结束后,哪个进程将获得 CPU 的使用权,继续执行。

6.4.3 do_IRQ 函数

在 Linux PowerPC 的外部中断处理子系统中,do_IRQ 函数作为总控程序,负责外部中断的处理。在 do_IRQ 函数之前的中断处理程序都是在为 do_IRQ 函数的运行作准备。该函数虽然使用 C 语言编写,但是并不易读。do_irq 函数的源代码在 ./arch/powerpc/kernel/irq.c 文件中。

进入 do_IRQ 函数时,E500 内核的 MSR 寄存器的值为 MSR_KERNEL。此时 E500 内核使能 Critical 中断和 Machine Check 中断。do_IRQ 函数的主要作用是执行设备驱动程序的中断服务例程,处理具体的中断事件。对于基于 E500 内核的 PowerPC 处理器,do_IRQ 函数可以简化为以下代码:

```
void do_IRQ(struct pt_regs *regs)
{
  struct pt_regs *old_regs = set_irq_regs(regs);
  unsigned int irq;
```

```
        irq_enter();

        /*
         * Every platform is required to implement ppc_md.get_irq.
         * This function will either return an irq number or -1 to
         * indicate there are no more pending.
         * The value -2 is for buggy hardware and means that this IRQ
         * has already been handled. -- Tom
         */
        irq = ppc_md.get_irq(regs);

        if (irq != NO_IRQ && irq != NO_IRQ_IGNORE) {
            generic_handle_irq(irq,regs);
        } else if (irq != NO_IRQ_IGNORE)
          /* That's not SMP safe... but who cares ? */
          ppc_spurious_interrupts++;

          irq_exit();
        set_irq_regs(old_regs);
} /* End do_IRQ */
```

do_IRQ 函数的执行流程如下：

- do_IRQ 函数首先使用 set_irq_regs 函数将 regs 参数保存起来。
- 之后调用宏 irq_enter 为外部中断处理作必要的准备。该宏调用 add_preempt_count 函数增加当前进程抢占计数 preempt_count 加上 HARDIRQ_OFFSET。
- 对于 MPC8541CDS 处理器系统，ppc_md.get_irq 函数等效与 mpic_get_irq 函数。所有使用 MPIC 中断处理程序的 PowerPC 处理器，其 ppc_md.get_irq 函数使用 mpic_get_irq 函数。ppc_md.get_irq 参数的赋值见本书的 8.2.5 节。
- 然后调用 generic_handle_irq 函数。该函数通过 mpic_get_irq 函数得到的软件中断号，之后调用设备驱动程序的中断服务例程。
- 调用宏 irq_exit 退出外部中断的处理。该函数的主要功能是调用 sub_preempt_count 函数减少抢占计数。在 irq_exit 函数中，还可能调用 invoke_softirq 函数，进行软中断的处理。
- 调用 set_irq_regs 将保存在 old_regs 中的值恢复。
- 当 do_IRQ 函数返回时，并不立即结束外部中断的处理，而是调用 ret_from_except 函数进行外部中断返回操作。

下文将详细说明在 do_IRQ 中出现的这些函数。其中 generic_handle_irq 函数是 Linux PowerPC 中断处理的主体。

1. 宏 irq_enter

宏 irq_enter 在 ./include/linux/hardirq.h 文件中，其源代码如下：

```
#define irq_enter()                                \
  do {                                             \
    account_system_vtime(current);                 \
    add_preempt_count(HARDIRQ_OFFSET);             \
    trace_hardirq_enter();                         \
  } while (0)
```

在宏 irq_enter 中，最重要的操作是将进程描述符的 preempt_count 参数增加 HARDIRQ_OFFSET。宏 irq_enter 这样做的目的是为了将来可以使用宏 hardirq_count 判断当前程序是否运行在中断处理程序的上下文中。

2. mpic_get_irq 函数

mpic_get_irq 函数的主要作用有以下两个：

- 通过读取 MPIC 中断控制器中的 IACK 寄存器启动外部中断响应周期。
- 将 IACK 寄存器中获得的硬件中断号转换为软件中断号。

mpic_get_irq 函数的源代码如下：

```
unsigned int mpic_get_irq(void)
{
  struct mpic *mpic = mpic_primary;

  BUG_ON(mpic == NULL);

  return mpic_get_one_irq(mpic);
}
```

mpic_get_irq 函数从 mpic_primary 变量中获得 mpic 结构，之后调用 mpic_get_one_irq 函数读取 IACK 寄存器，获得硬件中断号。之后，将此硬件中断号转换为软件中断号。mpic_get_one_irq 函数源代码如下所示：

```
unsigned int mpic_get_one_irq(struct mpic *mpic)
{
  u32 src;

  src = mpic_cpu_read(MPIC_INFO(CPU_INTACK)) &
MPIC_INFO(VECPRI_VECTOR_MASK);
#ifdef DEBUG_LOW
  DBG("%s:get_one_irq():%d\n",mpic->name,src);
#endif
  if (unlikely(src == MPIC_VEC_SPURRIOUS))
    return NO_IRQ;
  return irq_linear_revmap(mpic->irqhost,src);
}
```

- mpic_get_one_irq 函数首先从 MPIC 中断控制器的 IACK 寄存器中获得硬件中断号。

此时 src 变量保存着当前中断源的硬件中断号。

● 根据硬件中断号 src 调用 irq_linear_revmap 函数获得软件中断号。

irq_linear_revmap 函数的主要作用是从 irq_host 结构的反向中断映射表 revmap 中，获得与硬件中断号 src 对应的软件中断号。irq_linear_revmap 函数在 ./arch/powerpc/kernel/irq.c 文件中，其源代码的详细说明如下：

```
unsigned int irq_linear_revmap(struct irq_host * host,
        irq_hw_number_t hwirq)
{
  unsigned int * revmap;

  WARN_ON(host->revmap_type != IRQ_HOST_MAP_LINEAR);

  /* Check revmap bounds */
  if (unlikely(hwirq >= host->revmap_data.linear.size))
    return irq_find_mapping(host,hwirq);

  /* Check if revmap was allocated */
  revmap = host->revmap_data.linear.revmap;
  if (unlikely(revmap == NULL))
    return irq_find_mapping(host,hwirq);

  /* Fill up revmap with slow path if no mapping found */
  if (unlikely(revmap[hwirq] == NO_IRQ))
    revmap[hwirq] = irq_find_mapping(host,hwirq);

  return revmap[hwirq];
}
```

irq_linear_revmap 函数首先对 hwirq 参数进行检查，如果硬件中断号 hwirq 越界，则调用 irq_find_mapping 函数；如果反向中断映射表 revmap 为 NULL 时，也需要调用 irq_find_mapping 函数(在 Linux PowerPC 中，一般不会出现这两种情况)。

随后 irq_linear_revmap 函数检查当前硬件中断号是否已经在反向中断映射表 revmap 中建立与软件中断号的映射关系。

在 6.2.2 节中可知，在 Linux PowerPC 引导时，使用 irq_alloc_host 函数将中断映射表中的所有 Entry 都置为 IRQ_NONE。因此当外部中断处理函数第一次进入 do_IRQ 函数时，反向中断映射表 revmap 中与此硬件中断号对应的 Entry 一定为 IRQ_NONE。此时需要调用 irq_find_mapping 函数从中断映射表 irq_map 中获得软硬件中断号的映射关系，然后填入反向中断映射表 revmap 中。

此后，当此外部中断再次来临使 Linux PowerPC 进入 do_IRQ 函数时，将不会再执行 irq_find_mapping 函数，而是从反向中断映射表 revmap 中直接获得对应的软件中断号。Linux 系统使用反向中断映射表 revmap 的目的是在 do_IRQ 函数中，加速通过硬件中断号查找软

件中断号的速度。

3. generic_handle_irq 函数

generic_handle_irq 函数对外部中断进行真正意义上的处理。generic_handle_irq 函数的源代码在 ./include/linux/irq.h 文件中。generic_handle_irq 函数只有一个输入参数 irq,该参数表示当前中断源对应的软件中断号。该函数没有返回值,其源代码详解如下:

```
static inline void generic_handle_irq(unsigned int irq)
{
  struct irq_desc *desc = irq_desc + irq;

#ifdef CONFIG_GENERIC_HARDIRQS_NO__DO_IRQ
  desc->handle_irq(irq,desc);
#else
  if (likely(desc->handle_irq))
    desc->handle_irq(irq,desc);
  else
    __do_IRQ(irq);
#endif
}
```

generic_handle_irq 函数首先根据软件中断号 irq,在外部中断描述符表 irq_desc 中获得对应的外部中断描述符 desc。之后,该函数判断外部中断描述符 desc 的 handle_irq 参数是否为空。对于不同的外部中断源,其中断描述符 desc 的 handle_irq 参数有可能不为空,也有可能为空。

在基于 MPIC 中断控制器的 Linux 2.6 内核中,不同种类的外部设备将会为自己的中断描述符 desc 设置 handle_irq 参数。

例如,对于 MPC8541 处理器的 TSEC 控制器,desc->handle_irq 为 handle_fasteoi_irq 函数;而对于 MPC8541 处理器的外部中断源,其 desc->handle_irq 参数为空,此时需要调用 __do_IRQ 函数处理这些中断源。

__do_IRQ 函数的源代码在 ./kernel/irq/handle.c 文件中,对此有兴趣的读者可以自行阅读这段代码作为练习,本书对此将不做叙述。handle_fasteoi_irq 函数在 ./kernel/irq/chip.c 文件中。在 Linux 系统中中还有其他一些以 handle 开头的中断处理函数,这些函数也在 chip.c 文件中,如下:

- handle_simple_irq。此函数的实现与 handle_fasteoi_irq 函数几乎相同。
- handle_level_irq。此函数在处理结束时使用 mask 函数重新使能中断。8259 控制器使用了此函数处理外部中断。
- handle_edge_irq。此函数处理采用边沿触发的外部中断。在一个实际的系统中,采用边沿触发方式的外部中断较少。虽然 Linux PowerPC 定义了这个函数,但是这个函数不经常使用。
- handle_percpu_irq。这个函数主要用于基于多处理器的 Linux 系统,如 IBM 的 CELL 处理器。在 CELL 处理器中,外部中断被分为多种,有些中断必须由 CELL 处理器的某

个 CPU 单独处理，因此 Linux PowerPC 对此类中断的处理不需要使用自旋锁，该函数的处理过程也较为简单。

本书只介绍 handle _ fasteoi _ irq 函数，因为这个函数应用得最为广泛，其主要源代码和详细说明如下。handle _ fasteoi _ irq 函数共有 3 个输入参数。

- irq 参数用来描述软件中断号。
- desc 存放与外部中断对应中断描述符。
- regs 存放 PowerPC 处理器的系统寄存器。

```
void fastcall
handle_fasteoi_irq(unsigned int irq,struct irq_desc *desc)
{
  unsigned int cpu = smp_processor_id();
  struct irqaction *action;
  irqreturn_t action_ret;

  spin_lock(&desc->lock);

  if (unlikely(desc->status & IRQ_INPROGRESS))
    goto out;

  desc->status &= ~(IRQ_REPLAY | IRQ_WAITING);
  kstat_cpu(cpu).irqs[irq]++;

  /*
   * If its disabled or no action available
   * keep it masked and get out of here
   */
  action = desc->action;
  if (unlikely(!action || (desc->status & IRQ_DISABLED))) {
    desc->status |= IRQ_PENDING;
    goto out;
  }
```

handle _ fasteoi _ irq 函数需要对临界区 desc 进行访问，因此该函数需要获得自旋锁后才能访问 desc；之后，handle _ fasteoi _ irq 函数需要对存放在外部中断描述符 desc 内的中断状态，进行检查，该状态不能为 IRQ _ INPROGRESS；然后取出存放在中断描述符 desc 内的 action 指针，action 指针作为队首指针指向一个 irqaction 类型的队列。

action 指针队列的每一个 Entry，在设备驱动程序中，使用 request _ irq 函数进行中断申请时创建。在这个指针队列中，每一个 Entry 存放着相应外部中断服务例程的入口地址。

如果设备驱动程序在调用 request _ irq 函数时，不使用 IRQF _ SHARED 参数进行中断申请，那么在对应外部中断描述符 desc 的 action 指针队列中，只有一个数据成员；否则在对应外部中断描述符 desc 的 action 指针队列中，有一个或者多个数据成员。

```
    action_ret = handle_IRQ_event(irq,action);
    if (! noirqdebug)
      note_interrupt(irq,desc,action_ret);

    spin_lock(&desc->lock);
    desc->status &= ~IRQ_INPROGRESS;
out:
    desc->chip->eoi(irq);

    spin_unlock(&desc->lock);
}
```

这段程序首先调用 handle_IRQ_event 函数,处理 action 指针队列。handle_IRQ_event 函数的详细说明见 6.4.3 节。然后调用 desc->chip->eoi 函数,解除 PowerPC 处理器的“硬件中断状态”。对于基于 MPIC 中断控制器的 Linux PowerPC,desc->chip->eoi 函数等效于 mpic_end_irq 函数。

在 mpic_end_irq 函数中,调用 mpic_eoi 函数对 MPIC 中断控制器中的 EOI 寄存器进行写操作,解除 MPIC 中断控制器的中断状态。此时对于 PowerPC 处理器,外部中断已经处理完毕,但是对于 Linux PowerPC,中断处理过程并没有结束。

handle_fasteoi_irq 函数返回后,generic_handle_irq 函数将调用 irq_exit 函数,最后结束 do_IRQ 函数。这里再次提醒读者注意,外部中断处理程序在结束 do_IRQ 函数后,不会返回到被中断的程序,而是调用 ret_from_except 函数完成外部中断处理的返回。

4. handle_IRQ_event 函数

handle_IRQ_event 函数调用设备驱动程序的中断服务例程,对引发外部中断事件的中断源进行处理,该函数在 ./kernel/irq/handle.c 文件中,其详细说明如下:

```
irqreturn_t handle_IRQ_event(unsigned int irq,struct irqaction * action)
{
    irqreturn_t ret,retval = IRQ_NONE;
    unsigned int status = 0;

    handle_dynamic_tick(action);

    if (! (action->flags & IRQF_DISABLED))
      local_irq_enable_in_hardirq();
```

handle_IRQ_event 函数首先判断,相应的设备驱动程序在调用 request_irq 函数时,是否设置了 IRQF_DISABLED 位。如果该位被设置,则不调用 local_irq_enable_in_hardirq 函数使能外部中断,此时在整个外部中断处理过程中其他外部中断将无法重入。

如果设备驱动程序调用 request_irq 函数时没有设置 IRQF_DISABLED 位,则调用 local_irq_enable_in_hardirq 函数。local_irq_enable_in_hardirq 函数使用 wrteei 指令将 E500 内核的 MSR 寄存器 EE 位置为 1,使能外部中断。

在使能外部中断后，其他外部中断就有可能在当前中断没有处理完毕时，重入外部中断处理程序，从而形成中断嵌套。在一个系统中，占用处理器时间相对较长的中断处理函数最好不使用 IRQF_DISABLED 参数进行中断请求，以保证优先权较高的中断可以重入该中断服务例程，从而使优先权较高的中断可以获得更高的响应速度。

```
    do {
        ret = action->handler(irq,action->dev_id);
        if (ret == IRQ_HANDLED)
            status |= action->flags;
        retval |= ret;
        action = action->next;
    } while (action);

    if (status & IRQF_SAMPLE_RANDOM)
        add_interrupt_randomness(irq);
    local_irq_disable();

    return retval;
}
```

这段程序的执行流程如下：

(1) 这段程序调用 action->handler(irq,action->dev_id,regs)函数进行具体的外部中断处理。此函数是相应设备驱动程序在调用 request_irq 函数时，所注册的中断服务例程。

(2) 在此函数执行完毕后需要对返回值进行检查，如果返回值为 IRQ_HANDLED，则表示此函数已完成中断处理；否则表示没有完成中断处理，这种情况主要发生在 request_irq 函数使用 IRQF_SHARED 参数，进行中断申请的设备驱动程序中，此时可能有多个中断服务例程共享一个中断事件。如果发生的中断事件不由当前中断服务例程处理，则中断服务例程并不处理此中断事件，而直接退出，返回 IRQ_NONE，并由下一个中断服务例程处理该中断事件。

因此设备驱动程序的编写者，在编写共享中断事件的函数时，需要对当前中断事件进行判断。如果当前中断事件不是发向此中断事件处理函数，该函数需要立即返回，且返回值为 IRQ_NONE。

(3) 然后这段程序继续处理 action 指针队列的其他中断服务例程，直到 action 队列中所有的中断服务例程都处理完毕。

(4) 最后，这段函数调用 local_irq_disable 函数，禁止中断。由此可见，在 Linux PowerPC 外部中断处理系统中，只允许对在 local_irq_enable_in_hardirq 函数和 local_irq_disable 函数之间的代码进行重入。

5. irq_exit 函数

irq_exit 函数在 ./kernel/softirq.c 文件中定义。该函数的执行过程基本上是宏 irq_enter 执行的逆过程，其代码如下所示：

```
void irq_exit(void)
{
  account_system_vtime(current);
  trace_hardirq_exit();
  sub_preempt_count(IRQ_EXIT_OFFSET);
  if (! in_interrupt() && local_softirq_pending())
    invoke_softirq();
  preempt_enable_no_resched();
}
```

在 irq_exit 函数中，调用了一个重要的函数 invoke_softirq->do_softirq，该函数用来处理 Linux 系统中的软件中断，该函数同时满足以下两个执行条件时将被执行：

(1) in_interrupt 函数的返回值为 0，即当前函数没有在硬件中断或者软件中断的上下文中执行。在单 CPU 系统中，在 sub_preempt_count 函数执行完毕后，已经保证了该函数没有在硬件中断的上下文中，此处使用 in_interrupt 函数作为执行条件的主要目的是防止软件中断服务例程的重入。

(2) local_softirq_pending 函数的返回值为 1，即在当前 CPU 中存在未处理的软件中断。

此时 Linux 系统将调用 invoke_softirq 函数进行软件中断的处理，如果以上两个条件中有一个没有满足，则不进行软件中断的处理。

由于软件中断服务例程不可重入，因此如果 Linux 系统正在处理软件中断时，上下文被其他硬件中断程序切换，此时这个硬件中断程序将不能执行自己的软件中断程序，而是直接退出 irq_exit 函数。

Linux 系统采用这种机制可以保证软件中断程序能被 Linux 系统串行处理。

6.4.4 软件中断的处理

Linux 系统为了提高硬件中断处理程序的效率，引入了软件中断的概念。在 Linux 系统中，中断的处理可以被分为两个过程，分别是硬件中断处理与软件中断处理，这两个处理过程也经常被称作中断处理的上半部分与下半部分。

在 Linux 系统中，中断处理的上半部分指 invoke_softirq 函数之前的中断处理程序，而下半部分指 invoke_softirq 函数和 ksoftirqd 核心进程。

系统程序员在编写设备驱动程序时，可以将一些需要立即处理的中断服务程序放入中断处理的上半部分运行。使用 request_irq 函数挂接的中断服务例程运行在中断处理程序的上半部分。

有时，系统程序员还可以根据需要，将一些对实时性要求不高的程序进行延时处理。由上文得知，程序在中断处理的上半部分运行时，处理器处于中断状态；而程序在中断处理的下半部分运行时，处理器处于正常状态，中断和异常处理程序可以打断这段程序的运行，从而提高整个系统对中断的响应能力。

软件中断概念的引入目的就是减轻中断处理的负担，从而提高 Linux 系统对中断的响应能力。对于一个实际的应用，对驱动程序中的中断服务例程进行上下半的划分是一个系统工程，并没有一个统一的公式。本书不再阐述如何划分软硬件中断间的界限，只是提醒读者注意

这个划分对于 Linux 系统的实时性十分重要。

1. 软件中断处理程序的初始化

Linux 系统定义了多种软件中断类型，这些中断类型在 ./include/linux/interrupt.h 文件中定义，其含义如表 6-3 所示。

表 6-3 Linux 系统中的软件中断类型

软件中断类型	优 先 级	描 述
HI _ SOFTIRQ	0	级别最高的软件中断
TIMER _ SOFTIRQ	1	与系统时钟异常有关的软件中断
NET _ TX _ SOFTIRQ	2	与网卡发送数据报文相关的软件中断
NET _ RX _ SOFTIRQ	3	与网卡发送接收报文相关的软件中断
BLOCK _ SOFTIRQ	4	与硬盘交互相关的软件中断
TASKLET _ SOFTIRQ	5	常用的软件中断
SCHED _ SOFTIRQ	6	与 Linux SMP 的 Load Balance 算法有关的软件中断

优先级为 1～4 的软件中断类型是 Linux 系统分别为系统时钟、网络协议栈和硬盘特别设置的，普通设备驱动程序只能使用 HI _ SOFTIRQ 和 TASKLET _ SOFTIRQ 软件中断类型。其中 HI _ SOFTIRQ 软件中断类型用于一些对实时性要求较高的设备驱动程序，如声卡及显卡，而其他设备驱动程序使用 TASKLET _ SOFTIRQ 软件中断类型。

Linux 系统使用 soft _ irq _ int 函数将 HI _ SOFTIRQ 和 TASKLET _ SOFTIRQ 软件中断进行初始化，该函数的源代码在 ./kernel/softirq.c 文件中。

```
void __ init softirq _ init(void)
{
  open _ softirq(TASKLET _ SOFTIRQ, tasklet _ action, NULL);
  open _ softirq(HI _ SOFTIRQ, tasklet _ hi _ action, NULL);
}
```

softirq _ init 函数调用 open _ softirq 函数初始化相应的软件中断，并注册相应的软件中断处理函数。在 Linux 系统中，软件中断事件发生时，将调用相应的软件中断处理函数。

softirq _ init 函数的执行流程如下：

(1) 将 TASKLET _ SOFTIRQ 软件中断的处理函数设置为 tasklet _ action 函数。

(2) 将 HI _ SOFTIRQ 软件中断的处理函数设置为 tasklet _ hi _ action 函数。

在 Linux 系统中，NET _ TX _ SOFTIRQ 和 NET _ RX _ SOFTIRQ 软件中断类型由 ./net/core/dev.c 文件的 net _ dev _ init 函数进行注册；BLOCK _ SOFTIRQ 软件中断类型由 ./block/ll _ rw _ blk.c 文件的 blk _ dev _ init 函数进行注册；SCHED _ SOFTIRQ 软件中断类型由 ./kernel/sched.c 文件的 sched _ init 函数进行注册，其源代码如下：

```
/* net _ dev _ init 函数 */
   open _ softirq(NET _ TX _ SOFTIRQ, net _ tx _ action, NULL);
   open _ softirq(NET _ RX _ SOFTIRQ, net _ rx _ action, NULL);
```

```
/* blk_dev_init 函数 */
    open_softirq(BLOCK_SOFTIRQ,blk_done_softirq,NULL);

/* sched_init 函数 */
    open_softirq(SCHED_SOFTIRQ,run_rebalance_domains,NULL);
```

除此之外,Linux 系统还使用 do_pre_smp_initcalls->spawn_ksoftirqd->cpu_callback->kthread_create 函数初始化 ksoftirqd 核心进程。ksoftirqd 核心进程的主要作用是处理在 irq_exit 函数中因为软件中断不能被重入而未能及时处理的软件中断。ksoftirqd 函数在 ./kernel/softirq.c 文件中定义。其主要源代码如下所示:

```
static int ksoftirqd(void * __bind_cpu)
{
  set_user_nice(current,19);
  current->flags |= PF_NOFREEZE;
  set_current_state(TASK_INTERRUPTIBLE);
```

在 Linux SMP 中,每一个 CPU 都有自己的 ksoftirqd 进程。在 Linux SMP 中,使用"per_cpu(ksoftirqd,cpu)"函数调用相应的 ksoftirqd 核心进程。在 ksoftirqd 函数中,首先调用 set_user_nice 函数设置核心进程 ksoftirqd 的静态优先权设置为 139,由此可见该核心进程的优先权十分低;之后该函数将进程描述符的 flags 参数的 PF_NOFREEZE 位置为 1,表示该进程不可被挂起;最后该函数将该进程的状态设置为 TASK_INTERRUPTIBLE,准备将 ksoftirqd 进程切换出去。

```
while (! kthread_should_stop()) {
  preempt_disable();
  if (! local_softirq_pending()) {
    preempt_enable_no_resched();
    schedule();
    preempt_disable();
  }

  __set_current_state(TASK_RUNNING);
```

ksoftirq 函数的主体是一个大的 while 循环,只要 kthread_should_stop 函数的返回值为 0,该 while 循环将一直运行。在 while 循环中,ksoftirq 函数首先判断,在当前系统中,是否有未处理的软件中断。如果没有,ksoftirqd 进程将使用 schedule 函数将自己切换出去,使 ksoftirqd 进程进入等待状态,否则继续执行。当 ksoftirqd 进程被唤醒后,首先禁止进程抢占,之后将 ksoftirqd 进程的状态设置为 TASK_RUNNING。

```
while (local_softirq_pending()) {
        /* Preempt disable stops cpu going offline.
           If already offline, we'll be on wrong CPU:
           don't process */
```

```
            if (cpu_is_offline((long)__bind_cpu))
                goto wait_to_die;
            do_softirq();
            preempt_enable_no_resched();
            cond_resched();
            preempt_disable();
        }
        preempt_enable();
        set_current_state(TASK_INTERRUPTIBLE);
    }
    __set_current_state(TASK_RUNNING);
    return 0;
wait_to_die:
    preempt_enable();
    /* Wait for kthread_stop */
    set_current_state(TASK_INTERRUPTIBLE);
    while (!kthread_should_stop()) {
        schedule();
        set_current_state(TASK_INTERRUPTIBLE);
    }
    __set_current_state(TASK_RUNNING);
    return 0;
} /* End ksoftirqd */
```

这段程序再次判断当前系统是否有未处理的软件中断。在 Linux 系统中,虽然核心进程 ksoftirqd 的级别非常低,但是该进程仍然有可能在系统中没有任何软件中断事件时被激活运行。因此这段程序需要进行这种判断:如果有未处理的软件中断事件,则继续执行,否则将 ksoftirqd 进程的状态设置为 TASK_INTERRUPTIBLE,然后调用 schedule 函数将 ksoftirqd 进程切换出去。

(1) 调用 cpu_is_offline 函数,判断运行 ksoftirqd 进程的 CPU 是否在线,如果该 CPU 不在线,则跳转到 wait_to_die 标签处执行。

(2) 如果运行 ksoftirqd 进程的 CPU 在线,则调用 do_softirq 函数对 Linux 系统中未处理的软件中断一一处理。Linux 系统要求软件中断服务例程串行执行,因此当一个软件中断服务例程需要执行,发现另外一个软件中断服务例程没有执行完毕时,这个软件中断服务例程将不会执行。do_softirq 函数会串行地处理 Linux 系统中的软件中断,该函数的详细说明见下文。

(3) 允许进程抢占。将 ksoftirqd 进程的状态设置为 TASK_INTERRUPTIBLE,然后继续在"while (! kthread_should_stop())"循环中睡眠。

在正常情况下,核心进程 ksoftirqd 一经运行将不会结束,除非关闭 Linux 系统,或者由于系统异常,由 kthread_stop 函数结束核心进程 ksoftirqd。

2. 将软件中断服务例程挂接到软件中断处理程序

系统程序员在书写设备驱动程序时，可以将一些对实时性要求不高的中断服务例程，加入到 Linux 系统的软件中断处理程序中，之后由 do _ softirq 函数统一处理。Linux 系统提供了两个函数，tasklet _ schedule 和 tasklet _ hi _ schedule，实现这一功能。

其中 tasklet _ schedule 函数将 TASKLET _ SOFTIRQ 类型的软件中断服务例程加入到软件中断处理程序中；而 tasklet _ hi _ schedule 函数将 HI _ SOFTIRQ 类型的软件中断服务例程加入到软件中断处理程序中。

tasklet _ schedule 函数和 tasklet _ hi _ schedule 函数的实现机制较为类似，下文将详细介绍 tasklet _ schedule 函数的实现。设备驱动程序可以调用 tasklet _ schedule 函数将软件中断服务例程加入到软件中断处理程序中。

但是在此之前，系统程序员需要在设备驱动程序中，使用 tasklet _ init 函数对软件中断服务例程进行初始化。tasklet _ init 函数一共有三个输入参数，分别是 t、func 和 data 参数。其中 func 参数存放软件中断服务例程的地址，而 data 参数存放软件中断服务例程的输入参数。下文将重点介绍参数 t。

参数 t 是 tasklet _ struct 结构的变量。Linux 系统使用 tasklet _ struct 结构描述设备驱动程序中的软件中断服务例程，该结构在 ./include/linux/interrupt.h 文件中定义。该结构一共有 5 个数据成员，分别为 next，state，count，* func 和 data，其源代码如下所示：

```
struct tasklet _ struct
{
  struct tasklet _ struct  * next;
  unsigned long state;
  atomic _ t count;
  void ( * func)(unsigned long);
  unsigned long data;
};
```

(1) next 参数指向下一个可用的软件中断服务例程。在 Linux 系统中，所有类型相同的软件中断服务例程组成一个单向链表。

上文讲述过在 Linux 系统中，一共有 6 大类软件中断类型。为此在 Linux 系统中，一共设置了 6 个单向链表，连接所有类型相同的软件中断服务例程，存放 TASKLET _ SOFTIRQ 和 HI _ SOFTIRQ 类软件中断服务例程的链首指针在 ./kernel/softirq.c 文件中。

```
static DEFINE _ PER _ CPU(struct tasklet _ head, tasklet _ vec) = { NULL };
static DEFINE _ PER _ CPU(struct tasklet _ head, tasklet _ hi _ vec) = { NULL };
```

TASKLET _ SOFTIRQ 和 HI _ SOFTIRQ 软件中断的链首指针分别为 per _ cpu __ tasklet _ vec 和 per _ cpu __ tasklet _ hi _ vec，其数据类型为 tasklet _ head。

(2) state 参数表示软件中断服务例程的状态。state 参数为 TASKLET _ STATE _ SCHED 时，表示软件中断服务例程已经加入到软件中断队列中，等待处理；为 TASKLET _ STATE _ RUN 时，表示当前软件中断服务例程正在被处理。

(3) count 参数存放当前结构的原子锁。

(4) func 参数用来存放软件中断服务例程的入口地址。

(5) data 参数用来存放软件中断服务例程所需要的参数。

Linux 系统使用 tasklet _ init 函数，初始化软件中断服务例程的 tasklet _ struct 结构，该函数的源代码如下：

```
void tasklet_init(struct tasklet_struct * t,
        void ( * func)(unsigned long),unsigned long data)
{
    t->next = NULL;
    t->state = 0;
    atomic_set(&t->count,0);
    t->func = func;
    t->data = data;
}
```

该函数将依次设置 tasklet _ struct 结构的 next、state、count、func 和 data 参数。将 state 参数赋值为 0 等效于将此参数赋值为 TASKLET _ STATE _ SCHED。除了使用该函数，Linux 系统还可以使用宏 DECLARE _ TASKLET _ DISABLED 和 DECLARE _ TASKLET 静态创建 tasklet _ struct 结构，这两个宏的源代码如下：

```
#define DECLARE_TASKLET(name,func,data) \
struct tasklet_struct name = { NULL,0,ATOMIC_INIT(0),func,data }

#define DECLARE_TASKLET_DISABLED(name,func,data) \
struct tasklet_struct name = { NULL,0,ATOMIC_INIT(1),func,data }
```

当系统程序员需要实现软件中断服务例程时，必须首先将 tasklet _ struct 结构进行初始化，才能使用 tasklet _ schedule 函数将软件中断服务例程加入到软件中断处理程序中。

tasklet _ schedule 函数的定义在 ./include/linux/interrupt.h 文件中，其源代码如下。

```
static inline void tasklet_schedule(struct tasklet_struct * t)
{
    if (! test_and_set_bit(TASKLET_STATE_SCHED,&t->state))
        __tasklet_schedule(t);
}
```

tasklet _ schedule 函数首先检查 tasklet _ struct 结构之前存放的 TASKLET _ STATE _ SCHED 标示。注意 t->state 是一个临界变量，因此在本程序中使用原子操作 test _ and _ set _ bit 设置 TASKLET _ STATE _ SCHED 标示。

在这段程序中，test _ and _ set _ bit(TASKLET _ STATE _ SCHED, &t->state)函数的作用是将 TASKLET _ STATE _ SCHED 设置到 t->state 参数中，同时将 t->state 参数之前存放的数值返回。因此当 t->state 参数之前存放的数值为 TASKLET _ STATE _ RUN 时，即对应的软件中断服务例程正在被处理时，tasklet _ schedule 函数将直接返回；否则调用__ tasklet _ schedule 函数将当前软件中断服务例程加入到软件中断服务队列中。

__ tasklet _ schedule 函数在 ./kernel/softirq.c 文件中，其源代码如下所示：

```
void fastcall __ tasklet _ schedule(struct tasklet _ struct * t)
{
  unsigned long flags;

  local _ irq _ save(flags);
  t->next = __ get _ cpu _ var(tasklet _ vec).list;
  __ get _ cpu _ var(tasklet _ vec).list = t;
  raise _ softirq _ irqoff(TASKLET _ SOFTIRQ);
  local _ irq _ restore(flags);
}
```

3. do _ softirq 函数

当设备驱动程序使用 tasklet _ schedule 函数，将将软件中断服务例程挂接到软件中断处理程序后，Linux 系统使用 do _ softirq 函数处理这些软件中断服务例程。do _ softirq 函数在 ./arch/powerpc/kernel/irq.c 文件中定义，其源代码如下：

```
void do _ softirq(void)
{
  unsigned long flags;

  if (in _ interrupt())
    return;

  local _ irq _ save(flags);

  if (local _ softirq _ pending())
    do _ softirq _ onstack();

  local _ irq _ restore(flags);
}
```

如果当前进程运行在中断的上下文中，该函数将直接返回；否则关闭外部中断，之后进一步判断当前系统中是否有未处理的软件中断事件。如果在 Linux 系统中存在未处理的软件中断，则调用 do _ softirq _ onstack -> __ do _ softirq 函数处理此软件中断事件。

__ do _ soft _ irq 函数在 ./kernel/softirq.c 文件中定义，其源代码如下所示：

```
#define MAX _ SOFTIRQ _ RESTART 10

asmlinkage void __ do _ softirq(void)
{
  struct softirq _ action * h;
  __ u32 pending;
  int max _ restart = MAX _ SOFTIRQ _ RESTART;
```

```
    int cpu;

    pending = local_softirq_pending();
    account_system_vtime(current);
    __local_bh_disable((unsigned long)__builtin_return_address(0));
    account_system_vtime(current);

    __local_bh_disable((unsigned long)__builtin_return_address(0));
    trace_softirq_enter();

    cpu = smp_processor_id();
```

该函数首先获得当前 CPU 中的 pending 变量，然后调用__local_bh_disable 函数，最后调用 smp_processor_id 函数获得当前正在执行软件中断处理函数的 CPU。

__local_bh_disable 函数将增加当前进程的抢占计数为 SOFTIRQ_OFFSET，之后再使用 softirq_count 函数判断程序是否运行在软件中断的上下文中，返回值若为 1 表示当前程序运行在软件中断的上下文中。

```
restart:
    /* Reset the pending bitmask before enabling irqs */
    set_softirq_pending(0);

    local_irq_enable();
    h = softirq_vec;

    do {
        if (pending & 1) {
            h->action(h);
            rcu_bh_qsctr_inc(cpu);
        }
        h++;
        pending >>= 1;
    } while (pending);

    local_irq_disable();

    pending = local_softirq_pending();
    if (pending && --max_restart)
        goto restart;

    if (pending)
        wakeup_softirqd();
```

以上这段程序是 do _ softirq 函数的重要程序段。其执行流程如下：

(1) 这段程序首先清除当前 CPU 的 softirq 的 pending 位，以便 Linux 系统可以激活其他软件中断，然后使能外部中断。Linux 系统在软件中断的处理过程中，将在此使能外部中断以提高 Linux 系统的响应能力。提醒读者注意，这段程序必须首先清除 pending 位，然后再使能外部中断，以避免 Linux 系统死锁。

(2) 从 softirq _ vec 数组中获得软件中断服务队列中相应的 Entry 并存入到临时变量 h 中，TASKLET _ SOFTIRQ 类型的软件中断的操作函数为 tasklet _ action，而 HI _ SOFTIRQ 类型的软件中断的操作函数为 tasklet _ hi _ action。

因此在 Linux 系统中，存在未处理的 TASKLET _ SOFTIRQ 类型的软件中断时，将最终调用 tasklet _ action 函数，tasklet _ action 函数在 ./kernel/softirq.c 文件中定义。该函数将依次处理挂接在 per _ cpu __ tasklet _ vec 队列中的软件中断服务例程。

(3) 在“do-while(pending)”循环中，依次处理所有未处理的软件中断请求。

(4) 然后屏蔽外部中断，如果在处理软件中断服务例程时，又有新的软件中断产生，这段程序将跳转到 restart 标签处，重新运行。产生这种情况的原因是由于在这段程序的执行过程中，外部中断被使能，因此这段程序有可能被外部中断打断，从而产生了新的软件中断请求。

(5) 当这段程序已经运行了 max _ restart 次后(在 Linux 系统中 max _ restart 的缺省值为 10)，仍然有新的软件中断产生时，启动核心进程 ksoftirqd 处理新的软件中断请求，而不继续在__ do _ softirq 函数中处理这些软件中断。

```
        trace_softirq_exit();

        account_system_vtime(current);
        _local_bh_enable();
    } /* End __do_softirq */
```

_ local _ bh _ enable 函数是__ local _ bh _ disable 函数的逆过程，该函数执行完毕后，当前程序将退出软件中断程序的上下文。__ do _ softirq 函数结束后，do _ IRQ 函数将随之结束。此时 Linux 系统的外部中断处理程序将执行其最后一个步骤，从外部中断返回。外部中断的返回过程见 6.5 节。

6.4.5 工作队列 Work Queue

Linux 系统还提供了一种中断延时处理功能，即工作队列 Work Queue。Work Queue 与软件中断的实现思想类似，可以将某个工作推后执行。但是与软中断不同，采用工作队列机制可以将工作延时并交给一个核心进程来执行，因此在工作队列中允许进程进行重新调度及睡眠。而在软件中断中，不能够进行这些工作。

在系统设计中，可以自由地使用 Work Queue 或者软中断来实现中断的延时处理。如果一个延时任务在执行过程中需要睡眠，则必须选择 Work Queue 来实现，如果一个延时任务在执行过程中不需要睡眠，可以选择软中断来实现。Linux 系统使用 Work Queue 来处理一些信号机制，操作一些慢速的 I/O 设备。

如 Linux PowerPC 采用 Work Queue 实现 MPC8360 处理器千兆以太网控制器的 Phy 中断服务例程，这段程序在 ./drivers/net/ucc _ geth.c 文件中。

在使用 Work Queue 之前，首先需要将 Work Queue 中的数据成员，即某一个 Work 进行初始化。在这个驱动程序中，首先在 ucc _ geth _ open 函数调用宏 INIT _ WORK 初始化一个 Work——ugeth _ phy _ change，这段源代码如下：

```
/* Set up the PHY change work queue */
INIT_WORK(&ugeth->tq, ugeth_phy_change);
```

之后，当处理器系统出现 Phy 中断时，将在中断服务例程 phy _ interrupt 函数中，调用 schedule _ work 函数激活工作 ugeth _ phy _ change，之后 ugeth _ phy _ change 函数可以继续对 Phy 中断进行处理，但是 Phy 中断的服务例程可以很快结束。这段源代码如下：

```
static irqreturn_t phy_interrupt(int irq, void *dev_id)
{
...
  schedule_work(&ugeth->tq);
  return IRQ_HANDLED;
}
```

在这段代码中，可以发现，由于 Work Queue 的引入，千兆以太网控制器 Phy 中断的服务例程十分简单。系统程序员在编写该中断服务例程时，将处理该中断的主要工作放在 ugeth _ phy _ change 函数中，可极大程度地缩短 Phy 中断服务例程在中断上下文中的执行时间，从而提高 Linux 系统的中断处理效率。

ugeth _ phy _ change 函数的源代码如下，对此有兴趣的读者可以自行分析。

```
/* Scheduled by the phy_interrupt/timer to handle PHY changes */
static void ugeth_phy_change(struct work_struct *work)
{
  struct ucc_geth_private *ugeth =
    container_of(work, struct ucc_geth_private, tq);
  struct net_device *dev = ugeth->dev;
  struct ucc_geth *ug_regs;
  int result = 0;

  ugeth_vdbg("%s: IN", __FUNCTION__);

  ug_regs = ugeth->ug_regs;

  /* Delay to give the PHY a chance to change the
   * register state */
  msleep(1);

  /* Update the link, speed, duplex */
  result = ugeth->mii_info->phyinfo->read_status(ugeth->mii_info);
```

```
    /* Adjust the known status as long as the link
     * isn't still coming up */
    if ((0 == result) || (ugeth->mii_info->link == 0))
    adjust_link(dev);

    /* Reenable interrupts,if needed */
    if (ugeth->ug_info->board_flags & FSL_UGETH_BRD_HAS_PHY_INTR)
      mii_configure_phy_interrupt(ugeth->mii_info,
                MII_INTERRUPT_ENABLED);
}
```

1. kevent d_wq 工作线程的创建

由上文所述,采用工作队列机制,可以把工作延时并交给一个核心进程来执行。其中这个核心进程,如上文中的 ugeth_phy_change,是由 keventd_wq 工作线程创建的。

在 Linux 系统中,keventd_wq 工作线程由进程 1,即 init 进程初始化。init 进程调用 do_basic_setup->init_workqueues 函数初始化系统的 keventd_wq 工作线程。init_workqueues 函数的源代码如下:

```
void init_workqueues(void)
{
  singlethread_cpu = first_cpu(cpu_possible_map);
  hotcpu_notifier(workqueue_cpu_callback,0);
  keventd_wq = create_workqueue("events");
  BUG_ON(! keventd_wq);
}
```

init_workqueues 函数调用 create_workqueue->__create_workqueue 函数创建 keventd_wq 工作线程。__create_workqueue 函数除了缺省的工作线程 keventd_wq 之外,还可以创建自定义的工作线程,在有些设备驱动程序中,还可以使用 create_workqueue 函数创建一些自定义的工作线程。但是在大多数情况下,系统用户没有建立自定义工作线程的必要。

__create_workqueue 函数的源代码在 ./kernel/workqueue.c 文件中:

```
struct workqueue_struct *__create_workqueue(const char *name,
          int singlethread,int freezeable)
{
  int cpu,destroy = 0;
  struct workqueue_struct *wq;
  struct task_struct *p;

  wq = kzalloc(sizeof(*wq),GFP_KERNEL);
  if (! wq)
    return NULL;
```

在__create_workqueue 函数中,首先调用 kzalloc 函数申请工作队列使用的空间,Linux 系

统使用 workqueue_struct 结构表示一个工作队列，该结构的源代码如下：

```
struct workqueue_struct {
  struct cpu_workqueue_struct * cpu_wq;
  const char * name;
  struct list_head list;   /* Empty if single thread */
};
```

该结构的主要数据成员是 cpu_workqueue_struct 结构。在 Linux SMP 中，每一个 CPU 都有自己独立的 cpu_workqueue_struct 结构，该结构用来记录当前 Work Queue 的状态。name 参数用来记录当前工作队列的名称。list 参数将多个 workqueue_struct 结构连接成一个链表。在 Linux SMP 系统中，每一个 CPU 都有一个缺省的工作队列，这些工作队列通过 list 参数链接在一起，并使用全局指针 workqueues 指向这个链表的首部。

__create_workqueue 函数在申请工作队列使用的空间完毕后将执行以下程序。

```
wq->cpu_wq = alloc_percpu(struct cpu_workqueue_struct);
if (! wq->cpu_wq) {
  kfree(wq);
  return NULL;
}

wq->name = name;
```

在上述程序中，首先调用 alloc_percpu 函数分配工作队列使用的 cpu_workqueue_struct 结构。在 Linux SMP 中，需要将 cpu_workqueue_struct 结构存放到每一个 CPU 的 percpu_data 中，因此必须使用 alloc_percpu 函数申请 cpu_workqueue_struct 结构所用的空间。在基于单处理器的 Linux 系统中，alloc_percpu 函数等效于 kzalloc 函数。

之后这段程序对工作队列的 name 参数赋值。

```
mutex_lock(&workqueue_mutex);
if (singlethread) {
  INIT_LIST_HEAD(&wq->list);
  p = create_workqueue_thread(wq,singlethread_cpu,freezeable);
  if (! p)
    destroy = 1;
  else
    wake_up_process(p);
} else {
  list_add(&wq->list, &workqueues);
  for_each_online_cpu(cpu) {
    p = create_workqueue_thread(wq,cpu,freezeable);
    if (p) {
      kthread_bind(p,cpu);
      wake_up_process(p);
```

```
        } else
            destroy = 1;
        }
    }
    mutex_unlock(&workqueue_mutex);
```

当 singlethread 为 1 时，这段程序将使用 create_workqueue_thread 函数，为当前工作队列创建一个核心进程，然后再使用 wake_up_process 函数唤醒此核心进程。而 singlethread 为 0 时，这段程序为 Linux SMP 中的每一个 CPU 创建一个核心进程，并将这个核心进程与当前 CPU 绑定在一起，之后使用 wake_up_process 唤醒此核心进程。此后在系统中的每一个 CPU 都有一个核心进程处理这个工作队列。

```
    if (destroy) {
        destroy_workqueue(wq);
        wq = NULL;
    }
    return wq;
} /* __create_workqueue */
```

如果 create_workqueue_thread 函数执行失败，则调用 destroy_workqueue 函数释放这个工作队列占用的所有资源，然后进行函数返回。

在__create_workqueue 函数中，最重要的函数为 create_workqueue_thread 函数，该函数的主要作用是初始化每一个工作队列使用的 cpu_workqueue_struct 结构，同时创建工作队列使用的核心进程。该函数的源代码如下：

```
static struct task_struct *create_workqueue_thread(struct workqueue_struct *wq,
                int cpu,int freezeable)
{
    struct cpu_workqueue_struct *cwq = per_cpu_ptr(wq->cpu_wq,cpu);
    struct task_struct *p;

    spin_lock_init(&cwq->lock);
    cwq->wq = wq;
    cwq->thread = NULL;
    cwq->insert_sequence = 0;
    cwq->remove_sequence = 0;
    cwq->freezeable = freezeable;
    INIT_LIST_HEAD(&cwq->worklist);
    init_waitqueue_head(&cwq->more_work);
    init_waitqueue_head(&cwq->work_done);

    if (is_single_threaded(wq))
        p = kthread_create(worker_thread,cwq,"%s",wq->name);
```

```
    else
      p = kthread_create(worker_thread,cwq,"%s/%d",wq->name,cpu);
    if (IS_ERR(p))
      return NULL;
    cwq->thread = p;
    return p;
}
```

这段代码使用 kthread_create 函数创建了核心进程 worker_thread。worker_thread 进程用来处理 Linux 系统中的工作队列请求。

2. **设备驱动程序与 Work Queue 的挂接**

在下文中，我们还将以 MPC8360 处理器千兆网控制器的 Phy 中断服务例程说明设备驱动程序如何与 Work Queue 进行挂接。提醒读者注意，MPC8360 处理器的这个千兆网控制器使用 PQII Pro 芯片中的 QE 实现，而不是使用 TSEC 控制器实现。

在这个驱动程序中，首先定义一个 work_struct 结构的变量 tq，该变量保存在设备驱动程序的私有变量 ucc_geth_private 结构中，用来表示该设备驱动程序使用的 Work。ucc_geth_private 在 ucc_geth_probe 函数中被 alloc_etherdev 函数初始化。

```
struct ucc_geth_private {
...
  struct work_struct tq;
  struct timer_list phy_info_timer;
...
}
```

work_struct 结构用来描述一个 Work 的属性。该结构的源代码如下：

```
struct work_struct {
  atomic_long_t data;
#define WORK_STRUCT_PENDING 0      /* T if work item pending execution */
#define WORK_STRUCT_NOAUTOREL 1 /* F if work item automatically released on exec */
#define WORK_STRUCT_FLAG_MASK (3UL)
#define WORK_STRUCT_WQ_DATA_MASK (~WORK_STRUCT_FLAG_MASK)
  struct list_head entry;
  work_func_t func;
};
```

该结构有 3 个参数，其描述如下：

(1) data 参数为处理函数 func 的参数，该参数的最后两位，即 WORK_STRUCT_PENDING 和 WORK_STRUCT_NOAUTOREL，表示当前 Work 的状态。

当 WORK_STRUCT_PENDING 位为 1 时，表示当前 Work 正在等待处理；而 WORK_STRUCT_NOAUTOREL 位为 1 时，表示当前 Work 在处理结束后不会被自动释放。

(2) entry 参数将所有 work_struct 结构组成一个链表。在 Linux 系统中，每一个 CPU 为每种类型的 Work 维护一个链表。当与此 Work 相对应的工作线程被唤醒时，将执行这些链表

上的所有 Work。

(3) func 参数用来保存该 Work 所对应的处理函数。

在使用 tq Work 之前,千兆网控制器的驱动程序需要使用在 ucc_geth_open 函数中使用宏 INIT_WORK 初始化 tq Work,如上文所示。

宏 INIT_WORK 的源代码如下:

```
#define INIT_WORK(_work,_func)                                          \
  do {                                                                  \
    (_work)->data = (atomic_long_t) WORK_DATA_INIT(0);                 \
    INIT_LIST_HEAD(&(_work)->entry);                                    \
    PREPARE_WORK((_work),(_func));                                      \
  } while (0)
```

宏 INIT_WORK 将 work_struct 结构的 data、entry 和 func 参数分别赋值。在 ucc_geth.c 文件中,tq->func 参数为 ugeth_phy_change 函数。

宏 INIT_WORK 初始化 tq 变量后,ugeth_phy_change 函数将作为 tq Work 的处理函数。ugeth_phy_change 函数被 Phy 中断服务例程延时执行,并进一步对 Phy 中断进行相应处理。该函数运行在核心进程 worker_thread 的上下文中,因此该函数允许被外部中断,并且没有持有任何锁资源,在需要的时候,该函数还可以调用睡眠函数。

在 tq 变量初始化完毕后,设备驱动程序可以使用 schedule_work 函数调度这个 Work,也可以使用 schedule_delayed_work 函数延时一段时间后调度这个 Work。在大多数情况下,shedule_work 函数和 shedule_delayed_work 函数在设备驱动程序的中断服务例程中使用。在 ucc_geth.c 文件中,Phy 中断服务例程 phy_interrupt 函数中调用 schedule_work 函数激活 tq work。

shedule_work 函数调用 queue_work->__queue_work 函数将 tq work 加入到 keventd_wq 工作线程所在的工作队列中,然后激活 keventd_wq 工作线程中的 worker_thread 进程,在 worker_thread 进程被激活后,将会处理 tq Work。__queue_work 函数的源代码如下:

```
static void __queue_work(struct cpu_workqueue_struct *cwq,
        struct work_struct *work)
{
  unsigned long flags;

  spin_lock_irqsave(&cwq->lock,flags);
  set_wq_data(work,cwq);
  list_add_tail(&work->entry,&cwq->worklist);
  cwq->insert_sequence++;
  wake_up(&cwq->more_work);
  spin_unlock_irqrestore(&cwq->lock,flags);
}
```

在__queue_work 函数中,"wake_up(&cwq->more_work)"语句将激活核心进程 worker_thread。为充分理解这一过程,我们需要理解 worker_thread 进程的执行过程。

3. 核心进程 worker _ thread

在本节中，我们发现核心进程 worker _ thread 在 keventd _ wq 工作线程初始化时，由 create _ workqueue -> __ create _ workqueue 函数创建。

核心进程 worker _ thread 的源代码在 ./kernel/workqueue.c 文件中，其源代码详解如下：

```
static int worker_thread(void *__cwq)
{
  struct cpu_workqueue_struct *cwq = __cwq;
  DECLARE_WAITQUEUE(wait, current);
  struct k_sigaction sa;
  sigset_t blocked;

  if (!cwq->freezeable)
    current->flags |= PF_NOFREEZE;

  set_user_nice(current, -5);
```

这段程序首先定义一个等待事件 wait，然后将当前进程加入到这个等待事件中。如果当前工作队列的 freezeable 参数为 0，则将当前进程的 flags 参数置为 PF _ NOFREEZE，表示当前进程，即 worker _ thread 进程不能被挂起，之后将 worker _ thread 进程的静态优先权设置为 120-nice，即为 115。

```
...
  set_current_state(TASK_INTERRUPTIBLE);
  while (!kthread_should_stop()) {
    if (cwq->freezeable)
      try_to_freeze();

    add_wait_queue(&cwq->more_work, &wait);
    if (list_empty(&cwq->worklist))
      schedule();
    else
      __set_current_state(TASK_RUNNING);
    remove_wait_queue(&cwq->more_work, &wait);

    if (!list_empty(&cwq->worklist))
      run_workqueue(cwq);
    set_current_state(TASK_INTERRUPTIBLE);
  }
  __set_current_state(TASK_RUNNING);
  return 0;
}
```

这段程序是核心进程 worker _ thread 的程序主体。其执行顺序如下：

(1) 首先将当前进程的状态设置为 TASK _ INTERRUPTIBLE。

(2) 进入 while 循环,将当前进程加入到等待队列 cwq->more _ work 中。

(3) 如果当前 Work Queue 为空,当前进程调用 schedule 函数进入等待状态,否则当前进程的状态被重新设置为 TASK _ RUNNING 状态。

在 Linux 系统初始化时,当前 Work Queue 为空,所以 worker _ thread 函数将首先执行 schedule 函数,使当前进程在 wait 队列中等待。当设备驱动程序使用 schedule _ work 函数激活某个 Work 时,将调用 wake _ up(&cwq->more _ work)函数,唤醒在 cwq->more _ work 队列上睡眠的进程。

(4) 当 worker _ thread 进程被唤醒后,首先将当前进程从等待队列 cwq->more _ work 中摘除。

(5) 如果当前 Work Queue 不为空,则调用 run _ workqueue 函数处理当前 Work Queue 中的各个 Work。

(6) run _ workqueue 函数执行完毕后,当前进程的状态被设置为 TASK _ INTERRUPTIBLE,然后跳转回 while 循环。

run _ workqueue 函数将依次处理在当前 Work Queue 中所有的 Work,该函数的核心源代码如下所示。

```
static void run_workqueue(struct cpu_workqueue_struct *cwq)
{
...
  while (!list_empty(&cwq->worklist)) {
    struct work_struct *work = list_entry(cwq->worklist.next,
                struct work_struct,entry);
    work_func_t f = work->func;

    list_del_init(cwq->worklist.next);
    spin_unlock_irqrestore(&cwq->lock,flags);

    BUG_ON(get_wq_data(work) != cwq);
    if (!test_bit(WORK_STRUCT_NOAUTOREL,work_data_bits(work)))
      work_release(work);
    f(work);

    if (unlikely(in_atomic() || lockdep_depth(current) > 0)) {
...
    }

    spin_lock_irqsave(&cwq->lock,flags);
    cwq->remove_sequence++;
    wake_up(&cwq->work_done);
  }
```

```
        cwq->run_depth--;
        spin_unlock_irqrestore(&cwq->lock,flags);
    }
```

这段程序使用 while 循环，遍历当前 Work Queue 中的每一个 Work，并执行 Work 的 func 函数，最后调用 wake_up 函数唤醒在等待队列 cwq->work_done 中睡眠的进程。

在 Linux 系统中除了可以使用 schedule_work 函数和 schedule_delayed_work 函数对 Work 进行调度之外，还支持一个特殊的 Work 调度函数 flush_scheduled_work。该函数需要等待当前 Work Queueu 中的前一个 Work 执行完毕后才能执行。如果读者充分理解了以上 Work Queue 的全部内容，那么理解 flush_scheduled_work 函数是水到渠成的，本书不再介绍这个函数，读者可以自行阅读这个函数的源代码，作为练习。

6.5 外部中断的返回

do_IRQ 函数返回时，并不结束外部中断处理，而是调用 ret_from_except 函数进行外部中断返回。在 Linux PowerPC 外部中断进行返回时，需要对被中断进程的状态进行判断，然后决定是返回被中断进程处继续执行，还是执行其他优先权更高的进程。基于 E500 内核的 Linux PowerPC，外部中断返回函数 ret_from_except 在 ./arch/powerpc/kernel/entry_32.S 文件中。ret_from_except 函数将完成外部中断的恢复，该函数根据被中断程序是来自 Linux PowerPC 内核，还是应用程序，当前 Linux PowerPC 是否支持抢占式内核(preemption)，其执行过程有些不同。该函数执行的流程如图 6-3 所示。

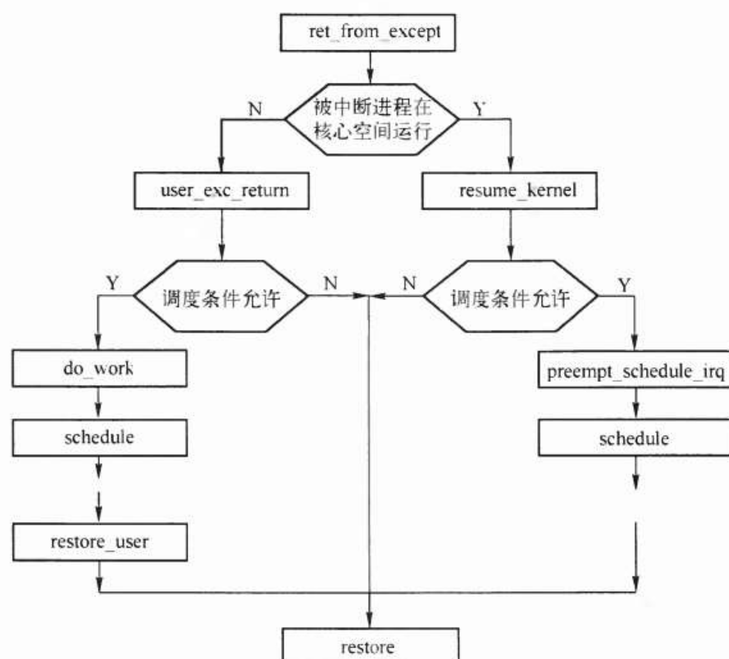


图 6-3 ret_from_except 函数的处理流程

由图 6-3 可知，在 Linux PowerPC 中，从应用进程进入到中断状态、完成中断处理、进行中断返回时，中断程序并不一定返回到该应用程序，而是调用 schedule 函数选择最适合的进程进行运行；从核心态进入中断程序后，进程中断返回时，将根据当前内核是否支持抢占式内核决定返回到哪里执行。

如果 Linux PowerPC 不支持内核抢占，中断将返回到 Linux PowerPC 核心，继续执行被中断的进程；如果 Linux PowerPC 支持抢占式内核，则根据被中断进程的状态决定是否调用 schedule 函数进行进程调度。

Linux PowerPC 使用汇编语言实现 ret _ from _ except 函数，其源代码详解如下：

```
/* ret _ from _ except 函数源代码片段 1 */
  .globl  ret _ from _ except
ret _ from _ except:
  /* Hard-disable interrupts so that current _ thread _ info()->flags
   * can't change between when we test it and when we return
   * from the interrupt. */
  LOAD _ MSR _ KERNEL(r10,MSR _ KERNEL)
  SYNC      /* Some chip revs have problems here... */
  MTMSRD(r10) /* disable interrupts */

  lwz r3,_ MSR(r1) /* Returning to user mode? */
  andi. r0,r3,MSR _ PR
  beq resume _ kernel

user _ exc _ return: /* r10 contains MSR _ KERNEL here */
```

这段程序的执行流程如下：

(1) 调用 LOAD _ MSR _ KERNEL 宏将 r10 赋值为 MSR _ KERNEL，在 E500 内核中，MSR _ KERNEL 为 MSR _ ME|MSR _ RI|MSR _ CE，即使能 Machine Check 中断和 Critical Interrupt，E500 内核不支持 RI 位。

(2) 这段程序将寄存器 r10 的值存放到 E500 内核的 MSR 寄存器中，此时 MSR 寄存器中的许多位将被屏蔽，包括 EE 位、DS 位和 IS 位等。

(3) 将存放在中断堆栈中的 MSR 寄存器赋值到 r3 寄存器中，并根据寄存器 r3 的值判断被中断的进程是运行在 Linux 核心空间还是用户空间。如果 PR 位为 1 表示被中断的进程运行在 Linux 用户空间，否则表示被中断的进程运行在 Linux 核心空间。

(4) 如果此进程运行在核心空间，则调用 resume _ kernel 函数，否则调用 user _ exc _ return 函数。

6.5.1 被中断进程运行在核心空间

当被中断进程运行在核心空间时，将执行 resume _ kernel 函数。此时，Linux PowerPC 将根据当前 Linux 内核是否使能内核抢占，分别处理。如果 Linux PowerPC 不使能内核抢占，Linux PowerPC 在中断返回后，将返回到被中断进程继续执行；否则有可能执行比被中断进程

优先级别高的进程。

resume _ kernel 函数源代码详解如下：

```
/* ret _ from _ except 函数源代码片段 2 */
#ifdef CONFIG _ PREEMPT
  b restore

/* N.B. the only way to get here is from the beq following ret _ from _ except. */
resume _ kernel:
  /* check current _ thread _ info->preempt _ count */
  rlwinm r9,r1,0,0,(31-THREAD _ SHIFT)
  lwz r0,TI _ PREEMPT(r9)
  cmpwi 0,r0,0      /* if non-zero,just restore regs and return */
  bne restore
  lwz r0,TI _ FLAGS(r9)
  andi. r0,r0,_ TIF _ NEED _ RESCHED
  beq+ restore
  andi. r0,r3,MSR _ EE  /* interrupts off? */
  beq restore      /* don't schedule if so */
1:bl preempt _ schedule _ irq
```

如果 Linux PowerPC 使能内核抢占，将执行以下代码：

(1) 从寄存器 r1 获得被中断进程的 thread _ info 的指针(由上文可知，进程的 thread _ info 与进程的核心堆栈共享一个 8KB 大小的空间，thread _ info 的内容存放在 8KB 空间的开始部分)，并将此指针放入 r9 寄存器中。

(2) 对 thread _ info-> preempt _ count 参数进行判断，如果 preempt _ count 参数不为 0，则表示当前进程不可被抢占。此时调用 restore 函数恢复被中断进程的现场空间；如果此参数为 0，表示该进程可以被抢占，继续执行。

(3) 对 thread _ info->flags 参数进行判断，如果 flags 参数中的_ TIF _ NEED _ RESCHED 位不为 1，即当前进程不需要进行调度时，将调用 restore 函数恢复被中断进程的现场空间；否则继续执行。

(4) 判断寄存器 r3(r3 保存着中断堆栈中的 MSR 寄存器，即进入中断处理程序之前 MSR 寄存器的值)中的 EE 位是否为 1，即外部中断是否使能。如果没有使能外部中断，则调用 restore 函数恢复被中断进程的现场空间，否则执行 preempt _ schedule _ irq 函数。

在 Linux PowerPC 中，有些异常处理程序，如缺页异常、系统调用等也调用了 ret _ from _ except 函数进行异常返回。这些异常处理结束后需要立即返回，因此在此，需要判断 EE 位是否使能。在缺页异常和系统调用中，MSR 寄存器的 EE 位一定为 0，因此在这些异常的处理程序中调用 ret _ from _ except 函数时，需要对 EE 位进行判断。

(5) preempt _ schedule _ irq 函数的源代码在 ./kernel/sched.c 文件中。在发生内核抢占时，该函数将调用 schedule 函数进行进程调度，使优先权较高的进程获得 CPU 资源。本书不再详细描述该函数的实现过程。

preempt _ schedule _ irq 函数执行完毕后，Linux PowerPC 将执行以下代码：

```
/* ret_from_except 函数源代码片段 3 */
  rlwinm r9,r1,0,0,(31-THREAD_SHIFT)
  lwz  r3,TI_FLAGS(r9)
  andi. r0,r3,_TIF_NEED_RESCHED
  bne-  1b
  /* interrupts are hard-disabled at this point */
restore:
```

这里提醒读者注意,preempt_schedule_irq 函数执行 schedule 函数进行进程调度,在一个 Linux PowerPC 系统中,也许在很长时间之后,才可能执行以上源代码。

这段代码首先从中断堆栈中获得被中断进程的 thread_info 指针,并将此值放入到寄存器 r9 中,之后判断 thread_info->flag 的 TIF_NEED_RESCHED 位是否为 1。如果为 1 需要继续调用 preempt_schedule_irq 函数进行调度,否则调用 restore 函数。restore 函数的详细解释如下所示:

```
/* ret_from_except 函数源代码片段 4 */
  /* interrupts are hard-disabled at this point */
restore:
  lwz  r0,GPR0(r1)
  lwz  r2,GPR2(r1)
  REST_4GPRS(3,r1)
  REST_2GPRS(7,r1)

  lwz r10,_XER(r1)
  lwz r11,_CTR(r1)
  mtspr SPRN_XER,r10
  mtctr r11

  PPC405_ERR77(0,r1)
  stwcx.  r0,0,r1              /* to clear the reservation */

  lwz r11,_LINK(r1)
  mtlr r11
  lwz r10,_CCR(r1)
  mtcrf 0xff,r10
  REST_2GPRS(9,r1)
  .globl exc_exit_restart
exc_exit_restart:
  lwz r11,_NIP(r1)
  lwz r12,_MSR(r1)
```

这段程序的主要作用是恢复 E500 内核的通用寄存器、XER、CTR、XER、LK 寄存器等,并清除 Reserve 位。

```
/* ret_from_except 函数源代码片段 5 */
exc_exit_start:
  mtspr SPRN_SRR0,r11
  mtspr SPRN_SRR1,r12
  REST_2GPRS(11,r1)
  lwz   r1,GPR1(r1)
  .globl exc_exit_restart_end
exc_exit_restart_end:
  PPC405_ERR77_SYNC
  rfi
  b.            /* prevent prefetch past rfi */
```

这段程序将保存在中断堆栈中的寄存器 SRR0 和 SRR1 恢复，并调用 rfi 指令最终退出外部中断处理，完成整个外部中断处理程序。这里有两个问题值得注意：

(1) rfi 指令的执行分为两个步骤，一是将 SRR1 寄存器中的值分别存放到 MSR 寄存器中；二是将程序调转到 SRR0 寄存器指定的位置处运行，同时进行指令同步。

在 Linux 系统中，跳转指令无法替代 rfi 指令，因为跳转指令无法实现在跳转的同时进行指令同步。在目前的 E500 内核版本中，如果用户改变 MSR 寄存器的 IS 位时，必须首先使用 rfi 指令将 SRR1 寄存器的值存放到 MSR 寄存器。这是因为在当前 E500 内核中有一个 Bug。如果程序员使用 mtspr 指令改写 MSR 寄存器的 IS 位，在指令 Cache 中有可能产生一个重复的 Cache 行。详情请参考 MPC85XX 勘误表中的“CPU29”。

(2) rfi 指令后“b.”指令的作用。在 Linux PowerPC 中，rfi 指令后一般会紧跟一个“b.”指令。这条指令将防止由于 E500 内核的指令预取而导致的一些不可预料的错误。

在绝大多数情况下，这种情况不会发生，因为 rfi 指令一般在“b.”指令之前执行，而 rfi 指令执行完毕后将进行指令同步，清空所有预取的指令后，进行跳转。

但是在某些情况下(发生这种情况的概率微乎其微，在一个正常的处理过程中，也许根本不会发生这种情况)，“b.”指令先于 rfi 指令执行。如果我们假设 rfi 指令后面的指令不是“b.”指令，而是其他指令如“lwz   r0,GPR0(r1)”，此时在 rfi 指令进行跳转之前，寄存器 r0 的值将被更改，从而影响程序的正常运行。而“b.”指令即使被执行，也不会对程序的正常运行带来影响。

把“b.”指令替换为“nop”指令也是一种选择。但是由于不同的处理器内核其指令预取部件的缓冲并不相同，因此程序员很难确定在 rfi 指令后，需要增加多少个“nop”指令。而采用“b.”指令可以合理地解决这一问题。执行“b.”指令时，将根据跳转指令的目标地址进行程序预取，而在跳转指令“b.”的目标地址(即当前地址)中，包含的指令依然为“b.”指令。因此在 rfi 指令后，仅需要紧跟一条“b.”指令就足够了。

6.5.2 被中断进程运行在用户空间

当被中断进程运行在用户空间时，外部中断处理程序将调用 user_exc_return 函数，该函数的详细说明如下：

```
/* ret _ from _ except 函数源代码片段 6 */
user _ exc _ return:      /* r10 contains MSR _ KERNEL here */
   /* Check current _ thread _ info()->flags */
  rlwinm r9,r1,0,0,(31-THREAD _ SHIFT)
  lwz  r9,TI _ FLAGS(r9)
  andi. r0,r9,(_TIF _ SIGPENDING|_TIF _ RESTORE _ SIGMASK|_TIF _ NEED _ RESCHED)
  bne  do _ work

restore _ user:
  lwz     r0,THREAD+THREAD _ DBCR0(r2)
  andis.  r10,r0,DBCR0 _ IC@h
  bnel-   load _ dbcr0
  b   restore
```

这段程序首先获得被中断进程 thread _ info 指针,之后判断当前进程是否需要重新调度及当前进程中是否有未处理的信号事件。如果有,则调用 do _ work 函数,否则调用 restore 函数完成外部中断的处理。本书将不对 Linux PowerPC 的信号机制进行讨论。

do _ work 函数的处理流程如图 6-4 所示。

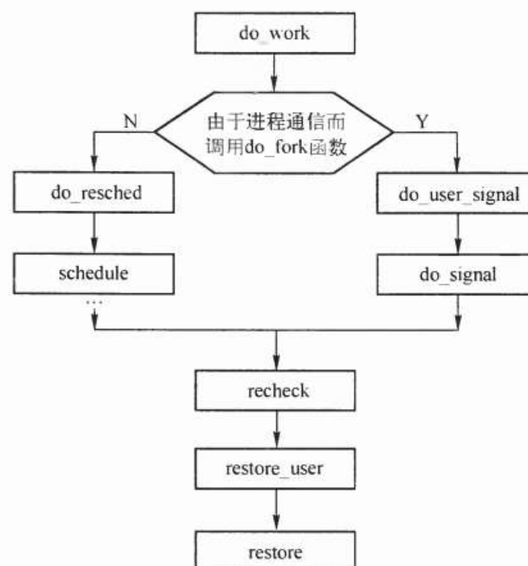


图 6-4　do _ work 函数的处理流程

do _ work 函数源代码的实现细节与 resume _ kernel 函数较为相似,本书对此不再叙述。希望读者可以对照以上流程图自行阅读这段代码,以充分地理解 Linux PowerPC 外部中断返回处理的所有细节。

6.6 Linux PPC 中的 OpenPIC 中断处理程序

Linux PPC 没有使用 MPIC 中断处理程序,而是使用 OpenPIC 中断处理程序。本文将简

单介绍 OpenPIC 中断处理程序中的主要数据结构与操作函数,并不会对此进行详细分析,因为这部分代码与 MPIC 中断处理程序相比,较为简单。在 Linux PPC 中,设置了许多数据结构,以及基于这些数据结构的操作函数,支持 Linux PPC 中的 OpenPIC 构架。

6.6.1 OpenPIC 中断处理程序的主要数据结构

OpenPIC 中断处理程序的主要数据结构在 ./arch/ppc/syslib/open_pic_defs.h 文件中。在 OpenPIC 中断控制器中,寄存器使用 128 位对界,以方便地支持 32 位、64 位及 128 位处理器。Linux PowerPC 使用以下结构对 OpenPIC 中的寄存器进行定义以确保这些寄存器 128 位对界:

```
typedef struct _OpenPIC_Reg {
  u_int Reg;
  char Pad[0xc];
} OpenPIC_Reg;
```

在 OpenPIC 中断控制器中,包含以下 3 类寄存器组:

(1) Global Register。此组寄存器描述 OpenPIC 中断控制器的公用信息。

(2) Interrupt Source Configuration Register。此组寄存器描述 OpenPIC 中断控制器的中断源。在 OpenPIC 构架中,最多支持 2048 个中断源。

(3) Per Processor Register。此组寄存器描述 OpenPIC 构架中的多个 CPU。在 OpenPIC 构架最多支持 32 个 CPU。

在 Linux PPC 中使用 OpenPIC 结构描述这些寄存器。OpenPIC 结构如下所示。

```
struct OpenPIC {
  char          Pad1[0x1000];
  OpenPIC_Global      Global;
  OpenPIC_Source      Source[OPENPIC_MAX_SOURCES];
  OpenPIC_Processor   Processor[OPENPIC_MAX_PROCESSORS];
};
```

在 OpenPIC 结构的开始处,使用了 Pad1 数据成员,将最开始的 0x1000 个字节保留。在 OpenPIC 控制器中,刚开始的 0x1000 字节为 OpenPIC_Processor 数据结构描述的寄存器,但是这些寄存器被称为 OpenPIC 控制器的私有寄存器,只能由处理器内部使用,而不能由用户通过指令进行访问。

OpenPIC_Global 结构包含了许多与 OpenPIC 中断控制器设置有关的寄存器以及和定时器有关的寄存器。OpenPIC_Global 结构的定义如下:

```
typedef struct _OpenPIC_Global {
  OpenPIC_Reg _Feature_Reporting0;
  OpenPIC_Reg _Feature_Reporting1;

  OpenPIC_Reg _Global_Configuration0;
```

```
    OpenPIC_Reg_Global_Configuration1;

    OpenPIC_Reg_Vendor_Specific[4];

    OpenPIC_Reg_Vendor_Identification;

    OpenPIC_Reg_Processor_Initialization;

    OpenPIC_Reg_IPI_Vector_Priority[OPENPIC_NUM_IPI];

    OpenPIC_Reg_Spurious_Vector;

    OpenPIC_Reg_Timer_Frequency;
    OpenPIC_Timer Timer[OPENPIC_NUM_TIMERS];
    char Pad1[0xee00];
} OpenPIC_Global;
```

在单处理器系统中，OpenPIC_Global结构中较重要的参数如下：

- _Feature_Reporting0参数包含当前OpenPIC的版本号，包括当前OpenPIC中一共支持多少个CPU，一共支持多少个中断源等一系列信息。
- 在_Global_Configuration0参数中，有两个重要的位R和P。对R位写1时，将复位OpenPIC中断控制器。P位为0时表示支持8259中断控制器的Pass Through方式（8259是一个常用的中断控制器），此时OpenPIC将工作在Pass Through方式下，在这种工作方式下，OpenPIC将所有自己管理的中断通过IRQ_OUT#引脚输出到8259中断控制器中，然后由8259统一管理所有的中断源。目前在嵌入式系统或者服务器系统中，很少使用这种方式设计中断系统。
- OpenPIC V1.3版本支持_Global_Configuration1参数。Linux PPC没有使用这个参数。IBM的MPIC中断控制器基于OpenPIC V1.2版本，因此在MPIC中断控制器中，也没有使用这个寄存器。
- _Processor_Initialization参数支持一个处理器对多个CPU的复位操作。但是在Linux PowerPC中，没有使用此参数进行多CPU系统的引导。Linux源代码的维护者似乎很不喜欢使用与其他体系结构不兼容的方式用来实现某种操作，即使有些方式对于某种体系结构的处理器十分有用。

OpenPIC_Global数据结构中的其他参数在Linux PowerPC中，或者没有实现，或者只对多处理器结构有效，本文不对此一一叙述。

在OpenPIC结构中，共定义了2048个OpenPIC_Source数据成员。这些数据成员用来对2048个Interrupt Source Registers进行描述，其结构如下所示：

```
typedef struct _OpenPIC_Source {
    OpenPIC_Reg_Vector_Priority;        /* Read/Write */
    OpenPIC_Reg_Destination;            /* Read/Write */
} OpenPIC_Source, *OpenPIC_SourcePtr;
```

- _Vector_Priority 参数识别 OpenPIC 中断控制器管理的中断源，及其优先权和中断触发方式(电平触发或者边沿触发)。
- _Destination 参数表示该中断由 OpenPIC 中断控制器中的哪个 CPU 处理。

在 OpenPIC 结构中，还定义了 32 个 OpenPIC_Processor 结构，该结构用来描述 OpenPIC 中断控制器支持的 CPU。这组寄存器包含一些通用的寄存器和多处理器系统用于中断交互的寄存器。

```
typedef struct _OpenPIC_Processor {
    u_int           IPI0_Dispatch_Shadow;                    /* Write Only */
    char            Pad1[0x4];
    u_int           IPI0_Vector_Priority_Shadow;             /* Read/Write */
    char            Pad2[0x34];
    OpenPIC_Reg     _IPI_Dispatch[OPENPIC_NUM_IPI];          /* Write Only */
    OpenPIC_Reg     _Current_Task_Priority;                  /* Read/Write */
    Char            Pad3[0x10];
    OpenPIC_Reg     _Interrupt_Acknowledge;                  /* Read Only */
    OpenPIC_Reg     _EOI;                                    /* Read/Write */
    char            Pad5[0xf40];
} OpenPIC_Processor;
```

在单处理器系统中，OpenPIC_Processor 数据结构中较为重要的参数如下所示：

- _Current_Task_Priority 参数。该参数用来描述 Current Task Priority 寄存器。此寄存器的低 4 位有效，可以表示 16 个当前进程的优先级。OpenPIC 规定只有当中断的优先级大于当前进程的优先级时中断才有可能传递到处理器。当前的 Linux 系统还使用此参数实现当前进程的优先级。
- _Interrupt_Acknowledge 参数。该参数存放中断响应寄存器(Interrupt Acknowledge Register)，该寄存器存放着当前待处理中断优先权最高的外部中断号。Linux PPC 使用 openpic_irq 函数读取中断响应寄存器，以获得当前外部中断号。
- _EOI 参数。该参数存放中断结束寄存器 EOI(End of Interrupt Register)。对该寄存器进行写操作时，优先权最高的外部中断被处理完毕。Linux PPC 使用 openpic_end_irq 函数或者 openpic_ack_irq 函数对该寄存器进行写操作，结束当前外部中断。

6.6.2 OpenPIC 中断处理程序的主要变量与操作函数

在 OpenPIC 中断处理程序中，主要的变量有 OpenPIC、OpenPIC_NumInitSenses、OpenPIC_InitSenses、NumProcessorsNumSources、open_pic_irq_offset 与 ISR，如表 6-4 所示。

表 6-4 OpenPIC 中断处理程序的主要变量

变 量 名	属 性	描 述
OpenPIC	volatile struct OpenPIC __iomem *	OpenPIC 寄存器基址的虚拟地址
OpenPIC_InitSenses	U_char *	保存处理器内部所有外部中断的触发方式
OpenPIC_NumInitSenses	U_int	以上描述符表的大小

（续）

变 量 名	属　　性	描　　述
NumProcessors	U _ int	OpenPIC 支持的 CPU 个数
NumSources	U _ int	OpenPIC 支持的外部中断个数
open _ pic _ irq _ offset	int	为 8259 中断控制器预留
ISR	volatile OpenPIC _ Source __ iomem *	ISR 指向在 OpenPIC 中，有效中断的个数。OpenPIC 支持 2048 个中断。

除了表 6-4 所示变量之外，OpenPIC 中断处理程序还设置了 open _ pic _ irq _ offset 变量，该变量的设置目的是为了给 8259 中断控制器预留中断号。

Linux PPC 将 0～open _ pic _ irq _ offset 之间的中断号预留给 8259 控制器。OpenPIC 中断控制器可以支持 2048 个中断，但是一个处理器系统中一般不会需要这么多的中断。在 Linux PPC 中，可以使用 OpenPIC _ Source 结构中的_ Vector _ Priority 参数动态地调整中断号。在使用 8259 这种低级中断控制器对外部中断进行管理时，外部中断号只能从 0 开始递增。因此 Linux PPC 中设立了 open _ pic _ irq _ offset 变量，将 0～open _ pic _ irq _ offset 之间的中断号预留给 8259 中断控制器。

在 OpenPIC 中断处理程序中，主要函数有 openpic _ enable _ irq、openpic _ disable _ irq、openpic _ ack _ irq、openpic _ end _ irq 和 openpic _ set _ affinity。这些函数如表 6-5 所示。

表 6-5　OpenPIC 中断处理程序的主要函数

函 数 名	描　　述
openpic _ enable _ irq	通过_ Vector _ Priority 参数将 OpenPIC 控制器指定的中断使能
openpic _ disable _ irq	通过_ Vector _ Priority 参数将 OpenPIC 控制器指定的中断禁止
openpic _ ack _ irq	OpenPIC 的中断响应函数。如果指定中断采用电平触发方式，此函数无意义，如果采用边沿触发方式，此函数可以对 EOI 寄存器进行写操作，结束当前中断处理
openpic _ end _ irq	OpenPIC 的中断结束函数，如果指定中断采用边沿触发方式，此函数无意义，如果采用电平触发方式，此函数将对 EOI 寄存器进行写操作，结束当前中断处理
openpic _ set _ affinity	设置外部中断的亲和性。在 Linux SMP 结构中，可以使用该函数确定由哪个处理器处理该外部中断

在 OpenPIC 结构中，还定义了一些 OpenPIC 结构初始化和如何利用 OpenPIC 结构进行中断处理的函数。Linux PPC 使用 OpenPIC 中断处理程序进行外部中断系统初始化时，会调用一个重要的函数 openpic _ init，此函数将逐个对 OpenPIC 中断控制器的相关寄存器初始化并对 irq _ desc 结构进行初始化。本书对 OpenPIC 中断处理程序不再作详细的介绍。在不久的将来，有关 OpenPIC 中断处理程序的源代码可能会被彻底移除。

6.7　Linux PowerPC 的系统调用

在 Linux 系统中，系统调用发挥着巨大的作用。如果没有系统调用，应用程序将不能访问 Linux 内核的任何资源。Linux 系统可以通过外部中断或者异常进入 Linux 核心空间，但是外部中断或者异常的发生是随机的，并不受程序员控制。而系统调用可以由程序员控制，使 CPU 主动地从应用空间切换到 Linux 内核空间。当系统调用运行结束后，CPU 又可以从 Lin-

ux 内核空间切换到应用空间。

Linux PowerPC 使用系统调用,将进程的用户空间和核心空间隔离。

进程运行在 Linux PowerPC 的用户空间时,只能访问 PowerPC 用户模式的寄存器,进行一些基本的逻辑运算操作。用户进程访问 Linux PowerPC 的外部设备,文件系统及网络系统等核心资源时,需要通过系统调用进入到 Linux 内核空间,获得这些资源,再将这些资源从 Linux 内核传递到用户空间。

Linux 系统这一机制有效保证了系统的核心空间与用户空间之间的相对独立性,某个进程在用户空间中的异常并不会影响 Linux 核心空间的稳定性。系统调用是 Linux 内核提供的一组功能强大的接口函数。Linux 系统对系统调用的支持从库函数开始。

在 Linux 系统中,需要使用系统调用的库函数,将在合适的位置调用 PowerPC 处理器提供的"sc 指令"。这条指令是 PowerPC 处理器为系统调用定制的,其他所有支持操作系统这种系统调用功能的处理器都有类似的这种指令,如 Intel i386 体系使用 INT 0x80 完成类似的功能。如果在处理器中没有这种指令,系统调用的功能将无法实现。

PowerPC 处理器运行 sc 指令后,将产生系统调用异常,使处理器从用户态切换到核心态。之后,Linux PowerPC 截获处理器产生的系统调用异常,调用内核提供的系统调用函数完成用户进程的系统调用请求。

Linux 系统共支持以下几类系统调用:

- 进程控制类系统调用。这类系统调用包括 fork、clone、execve、exit、_ exit、getpgid、setpgid、pause、nice 和 wait 等一系列系统调用。Linux PowerPC 提供了许多支持进程控制类的系统调用。但是应用软件程序员并不经常使用这类系统调用,其中许多系统调用并不被应用软件程序员熟悉。希望程序员有机会仔细浏览一遍这些系统调用。也许了解了这类系统调用后,应用软件程序员能够写出更加完备的应用程序。
- 文件系统控制类系统调用。这类系统调用共由两部分组成,第一部分是对文件操作的系统调用,第二部分是对文件系统操作的系统调用。文件操作类系统调用包括 open、create、close、write、read、fcntl、readv、writev、dup 和 dup2 等一系列系统调用。文件系统操作类系统调用包括 chmod、chown、chroot、access、mkdir、mknod、link 和 unlink 等一系列系统调用。
- 系统控制类系统调用。这类系统调用主要用来获取操作系统资源、控制文件读写和文件系统,包括 ioctl、acct、reboot、getitimer、setitimer、uname、time、create _ module、delete _ module 等一系列系统调用。
- 内存管理类系统调用,包括 brk、sbrk、mlock、mmap、munmap 和 mremap 等一系列系统调用。注意 Linux 系统并没有为应用软件程序员所熟悉的 malloc 函数,提供一条专门的系统调用。在多数情况下,用户调用 malloc 函数时,Linux 系统不会真正地进行物理内存申请。
- 网络管理类系统调用,包括 getdomainname、setdomainname、socketall、socket、connect、listen、send 和 receive 等一系列系统调用。
- 进程间通信类系统调用,包括 sigaction、signal、kill、sigsuspend、msgget、msgsnd 和 pipe 等一系列系统调用。

6.7.1 应用程序如何进入系统调用

在 Linux 系统中，与系统调用相关的库函数，其数量远大于 Linux 系统中，系统调用的数量，多个与系统调用有关的库函数可能共享同一个 Linux 系统中的系统调用。

在 Linux PowerPC 中，有关系统调用的宏定义在 ./include/asm-powerpc/unistd.h 文件中。

```
#define __NR_syscalls  302
#define NR_syscalls __NR_syscalls

#define __NR_restart_syscall    0
#define __NR_exit               1
#define __NR_fork               2
...
#define __NR_set_robust_list    300
#define __NR_move_pages         301
```

宏 NR_syscalls 存放系统调用的总数，在 Linux PowerPC 中，一共有 302 个系统调用。

在 Linux PowerPC 中，宏定义__NR_restart_syscall～__NR_move_pages 存放所有的系统调用号。如果用户希望自己添加一个系统调用，需要首先增加 NR_syscalls，然后再添加一个系统调用号，当然用户还需要在 Linux 内核中，添加相应的系统调用处理函数。一般情况下，Linux PowerPC 不建议用户采用添加新的系统调用。因为对于绝大多数应用，现有系统调用的数量已经足够。

Linux PowerPC 提供了一系列宏定义实现函数库与系统调用之间的接口，在这些接口宏中，将执行"sc"指令，从而引发 PowerPC 处理器的系统调用异常。Linux 2.6.20 源代码将这些接口宏从 Linux 源代码中移出，因为 Linux 系统的开发人员不认为这些代码属于 Linux 内核。但是在 Linux 2.6.19 的源代码中，依然可以在 ./include/asm-powerpc/unistd.h 文件中，发现这些宏。这些宏分别为_syscall0，_syscall1，_syscall2，_syscall3，_syscall4，_syscall5 和_syscall6，其源代码如下所示：

```
#define _syscall0(type,name)                              \
type name(void)                                           \
{                                                         \
    __syscall_nr(0,type,name);                            \
}

#define _syscall1(type,name,type1,arg1)                   \
type name(type1 arg1)                                     \
{                                                         \
    __syscall_nr(1,type,name,arg1);                       \
}
...
```

```
#define _syscall6(type,name,type1,arg1,type2,arg2,type3,arg3,          \
        type4,arg4,type5,arg5,type6,arg6)                              \
type name(type1 arg1,type2 arg2,type3 arg3,                            \
        type4 arg4,type5 arg5,type6 arg6)                              \
{                                                                      \
  __syscall_nr(6,type,name,arg1,arg2,arg3,arg4,arg5,arg6);             \
}
```

Linux 系统根据系统调用需要使用的参数将系统调用分成若干类，只有一个参数的系统通过宏_syscall1 进入 Linux 系统的系统调用处理程序；两个参数的系统调用通过宏_syscall2 进入 Linux 系统的系统调用处理程序，依此类推。目前在 Linux PowerPC 中，系统调用最多支持 6 个参数。

在_syscallx 宏中，type 参数存放对应参数的类型，name 参数表示系统调用的名字，arg 参数用来存放参数的名字。宏_syscallx 调用宏__syscall_nr 执行 sc 指令。库函数在使用系统调用时，首先需要使用宏_syscallx 对需要使用的系统调用进行原型定义，之后使用这些系统调用。下文以 time 系统调用为例，说明用户库如何使用_syscallx 宏。

```
#include <stdio.h>
...
_syscall1(time_t,time,time_t *,tloc)
main()
{
  time_t current_time;
  the_time=time((time_t *)0);
  printf("The time is %ld\n",current_time);
}
```

上述程序首先调用宏_syscall1 对 time 函数进行原型定义，之后调用 time 函数获得系统时间，并使用 printf 将此信息输出。_syscall1(time_t,time,time_t *,tloc) 等效于以下代码：

```
time_t time(time_t * tloc)
{
  __syscall_nr(1,time_t,time,tloc);
}
```

在 Linux PowerPC 中，宏_syscall_nr 执行系统调用指令 sc，完成用户空间与 Linux 内核空间的交互。宏_syscall_nr 源代码如下所示：

```
#define __syscall_nr(nr,type,name,args...)                  \
  unsigned long __sc_ret,__sc_err;                          \
  {                                                         \
    register unsigned long __sc_0 __asm__ ("r0");           \
    register unsigned long __sc_3 __asm__ ("r3");           \
    register unsigned long __sc_4 __asm__ ("r4");           \
```

```
register unsigned long __sc_5 __asm__ ("r5");          \
register unsigned long __sc_6 __asm__ ("r6");          \
register unsigned long __sc_7 __asm__ ("r7");          \
register unsigned long __sc_8 __asm__ ("r8");          \
                                                       \
__sc_loadargs_##nr(name,args);                         \
```

这段程序首先使用 r0 和 r3~8 寄存器，保存变量__sc_0，__sc_3~__sc_8。然后这段程序使用宏__sc_loadargs_将系统调用号__NR_##name 存放到__sc_0 中，即寄存器 r0 中。在 Linux 系统中，系统调用处理函数使用系统调用号，查找相应的系统调用处理函数。

```
__asm__ __volatile__                                   \
  ("sc          \n\t"                                  \
  "mfcr %0      "                                      \
  :"=&r" (__sc_0),                                     \
  "=&r" (__sc_3),"=&r" (__sc_4),                       \
  "=&r" (__sc_5),"=&r" (__sc_6),                       \
  "=&r" (__sc_7),"=&r" (__sc_8)                        \
  :__sc_asm_input_##nr                                 \
  :"cr0","ctr","memory",                               \
  "r9","r10","r11","r12");                             \
```

以上这段程序使用了 Gcc 的内联汇编代码，Gcc 的内联汇编代码可以将汇编语言直接嵌入到 C 代码中执行。初次接触这种汇编结构的用户，可能会对此不适应。但是采用这种内联汇编代码，可以由 Gcc 动态地分配通用寄存器资源，并可以直接使用在 C 程序中定义的变量。

这种格式的汇编语句使用“:”将整个代码分为四个部分。

- 第一部分是汇编代码本身，被称为指令部分，其格式与在 PowerPC 汇编语言基本相同，这一部分在第一个“:”之前。
- 第二部分在第一个“:”与第二个“:”之间，此部分是输出部分(Output)，用于描述输出操作数。其中，不同操作数之间需要用逗号格开，每个操作数描述符由 C 语言变量组成。在每个输出操作数的字符串中，必须包含“=”表示它是一个输出操作数。
- 第三部分在第二个“:”与第三个“:”之间，此部分是输入部分(input)，该部分表示输入操作数，不同的操作数之间必须用逗号格开。
- 最后一个部分在第三个“:”之后，此部分是破坏描述部分(Clobber/Modify)，通知 Gcc，当前内联汇编语句可能会对某些寄存器或内存进行修改，希望 Gcc 在生成汇编代码时能够将这些被破坏的部分进行预处理。

Gcc 内联汇编的语法和实现较为复杂，本书仅说明与宏定义__syscall_nr 有关的语法，对此有兴趣的读者可以参考 Gcc 使用手册，见 http://gcc.gnu.org/onlinedocs。Gcc 的使用手册详细讲述了如何使用内联汇编。

1. 宏__syscall_nr 内联代码的第一部分

宏__syscall_nr 内联代码的第一部分如下：

```
__asm____volatile__                                    \
  ("sc            \n\t"                                 \
  "mfcr %0       "                                      \
```

- "__asm__"表示其后的代码为 Gcc 的内联汇编,"__volatile__"表示编译器不要优化这段代码。
- "sc"是 PowerPC 处理器的系统调用指令,执行此指令后,PowerPC 处理器将产生系统调用异常,并由 Linux PowerPC 处理系统调用异常。"\n\t"是制表符,Gcc 内联汇编规定,除了最后一条汇编语句外,在其他每一条语句后,都必须加入"\n\t"制表符。
- 当系统调用异常处理函数返回时,执行 mfcr 指令将 CR 寄存器的内容存入%0 所代表的变量中。%0 用于描述第一个变量。

2. 宏__syscall_nr 内联代码的第二部分

宏__syscall_nr 内联代码的第二部分如下所示:

```
  : "=&r" (__sc_0),                                     \
    "=&r" (__sc_3),"=&r" (__sc_4),                      \
    "=&r" (__sc_5),"=&r" (__sc_6),                      \
    "=&r" (__sc_7),"=&r" (__sc_8)                       \
```

- __sc_0 代表%0,__sc_3 代表%1,__sc_4 代表%2……__sc_8 代表%6。
- 在这段代码中,"="前缀表示,它所修饰的变量(如__sc_0)是一个输出变量;"&"前缀表示,它所修饰的变量不能和输入变量使用相同的寄存器,"r"前缀表示,它所使用的变量将放入通用寄存器中。细心的读者可能已经发现在宏定义__syscall_nr 中,实际上已经定义了__sc_0、__sc_3~__sc_8 使用 r0、r3~r8 寄存器,因此在这里使用前缀"r"并没有太大意义。

3. 宏__syscall_nr 内联代码的第三部分

宏__syscall_nr 内联代码的第三部分如下:

```
  : __sc_asm_input_##nr                                 \
```

对于只有一个操作数的系统调用,nr 为 1,上述代码等效于以下代码:

```
  : "0" (__sc_0),"1" (__sc_3)                           \
```

- 在这条语句中,"0"和"1"前缀表示,__sc_0 使用%0 所代表的变量,而__sc_3 使用%1 所代表的变量。
- 对于只有一个操作数的系统调用,上述代码表示__sc_0 和__sc_3 是一个输入/输出变量,并且其使用的寄存器分别是 r0 和 r3。

4. 宏__syscall_nr 内联代码的第四部分

宏__syscall_nr 内联代码的第四部分如下:

```
  : "cr0","ctr","memory",                               \
    "r9","r10","r11","r12");                            \
```

内联汇编的第四部分声明了一些在汇编程序中使用的寄存器,这些寄存器需要被保存起

来以便恢复。"memory"的作用需要稍微解释一下,编译器在优化处理器有关内存访问的代码时,经常将内存变量保存到寄存器中,调整指令顺序,并充分利用处理器的指令流水线。在某些情况下,这种指令顺序的调整会引起内存一致性问题。Linux 系统可以设置内存屏障(memory barrier)解决这些问题。

这段内联汇编中的"memory"就是起内存屏障的作用。在内联汇编中如果有指令修改了内存,需要在第四部分增加"memory",通知 Gcc 编译器,内存可能会被修改,Gcc 得知这个信息后,就会在这段指令之前,插入必要的指令保证内存的一致性。

5. 宏__syscall_nr 在内联汇编代码之后的源代码

```
        __sc_ret = __sc_3;                              \
        __sc_err = __sc_0;                              \
    }                                                   \
    if (__sc_err & 0x10000000)                          \
    {                                                   \
        errno = __sc_ret;                               \
        __sc_ret = -1;                                  \
    }                                                   \
    return (type) __sc_ret
```

- 执行 sc 指令后,Linux PowerPC 将进行系统调用异常处理,其处理过程将在下文详细介绍。在执行完系统调用异常后,通用寄存器 r3 保存返回值,而通用寄存器 r0 保存错误状态。
- 如果系统调用返回时出现错误,则使用变量 errno 保存错误代码,并将变量__sc_ret 赋值为-1 表示有错误发生。

6.7.2 Linux PowerPC 的系统调用异常

执行 sc 指令时,PowerPC 处理器将产生系统调用异常。E500 内核使用 IVPR 与 IVOR8 寄存器,保存系统调用异常向量的有效地址。在执行 sc 指令后,PowerPC 处理器将进行以下操作。

(1) 将 sc 指令的下一条指令的有效地址存放到 SRR0 寄存器中。

(2) 将当前 MSR 寄存器的内容存放到 SRR1 寄存器中。

(3) 保留 MSR 寄存器的 CE、ME 和 DE 位,其他位清零。

(4) 将程序跳转到 IVPR 和 IVOR8 指定的地址处执行。

在 Linux PowerPC 中,系统调用与外部中断的初始化较为类似。Linux PowerPC 使用宏定义 SET_IVOR 将 PowerPC E500 内核中的 IVOR8 寄存器赋值为 SystemCall 函数的有效地址,其源代码如下:

```
SET_IVOR(8, SystemCall);
```

当系统调用异常发生后,Linux PowerPC 调用 SystemCall 函数对系统调用进行处理。SystemCall 函数的源代码在./arch/powerpc/kernel/head_fsl_booke.S 文件中。这段代码等效于以下源代码:

```
.align 5
SystemCall:
  NORMAL_EXCEPTION_PROLOG;
  EXC_XFER_EE_LITE (0x0c00,DoSyscall)
```

这段程序与 ExternalInput 函数的执行过程基本相同。只是在 SystemCall 的初始化中，使用了宏 EXC_XFER_EE_LITE 而不是 EXC_XFER_LITE。

这两个宏的主要区别在于，在 EXC_XFER_EE_LITE 宏中，使用宏 COPY_EE 将 MSR 寄存器的 EE 位置为 1；而在 EXC_XFER_LITE 中，将 MSR 寄存器的 EE 位置为 0。因此在 Linux PowerPC 处理系统调用异常时，外部中断将被使能，从而提高了整个 Linux 系统对外部中断的响应能力。

宏 NORMAL_EXCEPTION_PROLOG 的详细描述见 6.4.1 节。宏 EXC_XFER_EE_LITE (0x0c00,DoSyscall)等效于以下代码：

```
li  r10,0x501;                          \
stw  r10,TRAP(r11);                     \
lis  r10,MSR_KERNEL @h;                 \
ori  r10,r10,MSR_KERNEL @l;             \
COPY_EE (r10,r9)                        \
bl  transfer_to_handler;                \
.long  DoSyscall;                       \
.long  ret_from_except
```

这段代码与 ExternalInput 中断处理程序的相应代码类似。SystemCall 异常处理程序在执行完 transfer_to_handler 函数之后，分别调用 DoSyscall 和 ret_from_except 函数，进行 Linux 系统调用的处理。

其中 transfer_to_handler 和 ret_from_except 函数的处理过程与外部中断的处理过程相同。读者可以回顾本章有关外部中断处理的内容，了解这两个的函数的实现过程。

在 Linux PowerPC 中，DoSyscall 函数是处理系统调用异常的主要组成部分，其函数的源代码见 ./arch/powerpc/kernel/entry_32.S 文件。

DoSyscall 函数由两部分组成，第一部分是进入系统调用异常处理函数 DoSyscall，第二部分是从系统调用异常中返回 ret_from_except。

1. 系统调用异常处理函数

DoSyscall 函数的源代码如下：

```
_GLOBAL(DoSyscall)
  stw  r0,THREAD+LAST_SYSCALL(r2)
  stw  r3,ORIG_GPR3(r1)
  li  r12,0
  stw  r12,RESULT(r1)
  lwz  r11,_CCR(r1)  /* Clear SO bit in CR */
  rlwinm r11,r11,0,4,2
  stw  r11,_CCR(r1)
```

```
    rlwinm r10,r1,0,0,(31-THREAD_SHIFT) /* current_thread_info() */
    lwz  r11,TI_FLAGS(r10)
    andi. r11,r11,_TIF_SYSCALL_T_OR_A
    bne- syscall_dotrace
syscall_dotrace_cont:
    cmplwi  0,r0,NR_syscalls
    lis  r10,sys_call_table@h
    ori  r10,r10,sys_call_table@l
    slwi  r0,r0,2
    bge- 66f
    lwzx  r10,r10,r0 /* Fetch system call handler [ptr] */
    mtlr  r10
    addi  r9,r1,STACK_FRAME_OVERHEAD
    PPC440EP_ERR42
    blrl        /* Call handler */
```

在 Linux PowerPC 中，所有系统调用首先进入 DoSyscall 函数，之后根据系统调用号，在系统调用表中，查找到相应系统调用的处理函数后，执行此函数。这段程序的原理较为简单，请读者自行分析理解。

在 Linux PowerPC 中，使用 r0 存放系统调用号，使用 sys_call_table 变量存放系统调用处理函数的基地址，即系统调用表。sys_call_table 变量在 ./arch/powerpc/kernel/systbl.S 文件中定义，其代码如下所示：

```
_GLOBAL(sys_call_table)
#include <asm/systbl.h>
```

由此可见，系统调用表实际存放在 ./include/asm-powerpc/systbl.h 文件中，该文件存放着 Linux PowerPC 所有系统调用处理函数的入口地址，如下所示：

```
SYSCALL(restart_syscall)
SYSCALL(exit)
PPC_SYS(fork)
SYSCALL_SPU(read)
SYSCALL_SPU(write)
COMPAT_SYS_SPU(open)
...
SYSCALL_SPU(faccessat)
COMPAT_SYS_SPU(get_robust_list)
COMPAT_SYS_SPU(set_robust_list)
```

需要提醒读者注意，在这个文件中，系统调用处理函数的存放顺序需要与 ./include/asm-powerpc/unistd.h 文件中定义的系统调用号一致。

比如__NR_exit 的值为 1，其处理函数 sys_exit 必须排在 systbl.h 文件中的第二行。sys

_ restart _ syscall 函数排在 systbl.h 文件的第一行，为 sys _ call _ table 指针所指向的第一个系统调用处理函数。

下文以系统调用 open 为例，说明系统调用处理函数在 Linux PowerPC 中的执行过程。首先程序员使用 Glibc 中提供的 open 库函数进行系统调用操作。open 库函数有三个参数，分别为 pathname，oflag 和 mode。程序员也可以只使用前两个参数 pathname 和 oflag。open 函数的原型定义如下：

```
int open (const char * pathname, int oflag, .../* ,mode_t mode */);
```

open 函数的执行过程如下：

- 执行 open 函数时，Linux 系统调用_ syscall2(int,open,const char *,int,oflag)函数，并在_ syscall2 函数中，执行"sc"指令。
- 之后 Linux PowerPC 进入系统调用异常中断程序，此时寄存器 r3 和寄存器 r4 的值分别与 pathname 和 oflag 参数对应。而寄存器 r0 被赋值为 5。
- 通过 sys _ call _ table 查询，得知 COMPAT _ SYS _ SPU(open)函数将处理 open 系统调用，COMPAT _ SYS _ SPU(open)函数与 sys _ open 函数等效。sys _ open 函数的参数为 filename，flag，mode，与 open 库函数的参数一一对应。
- 执行 sys _ open 函数，在 Linux 内核中，创建文件描述符和相应的 inode 节点。最后从系统调用异常中返回。

2. 从系统调用异常中返回

DoSyscall 函数在"blrl"指令执行完毕后，从系统调用异常中返回。Linux PowerPC 在系统调用返回时，需要做以下工作：

(1) 检查系统调用处理函数返回状态，并根据返回状态更新 CR0。

(2) 恢复系统调用异常现场，跳转到 ret _ from _ except 函数，完成系统调用异常的返回。ret _ from _ except 函数的详细说明见 6.5 节。

第7章　Linux PowerPC的内存管理

存储器是处理器系统的重要组成部分,由内部存储器和外部存储器组成。其中,内部存储器,如SDRAM、DDR-SDRAM、SRAM等,用来保存程序使用的指令与数据,可以与处理器直接交换数据。而外部存储器,如Flash、硬盘等,用来存放程序或者其他静态的数据。

Linux系统使用文件系统管理外部存储区,使用内存管理子系统管理内部存储器。在Linux系统中,内存是最重要的资源。如果将进程调用子系统比喻为Linux系统的心脏,那么内存管理系统就是Linux系统的血液。在Linux系统中,内存管理子系统是最核心,也是最难理解的一部分内容。

Linux系统的内存管理主要包含以下内容:

(1) 虚实地址转换。在一个32位处理器系统中,物理内存空间最大为4 GB。在这个4 GB的物理空间中,还包括外部设备使用的一些物理地址空间,如PCI总线,PowerPC处理器内部的寄存器等。在Linux系统中,这个4 GB大小的物理地址空间最终要映射到Linux内核直接管理的1 GB核心空间中,即0xC000-000~0xFFFF-FFFF这段地址空间中。因此在Linux系统中需要进行物理地址空间到Linux内核的虚拟地址空间的转换。

除此之外,在32位处理器系统中,Linux系统的每一个用户进程,都有3 GB大小的虚拟地址空间,而且每一个用户进程都使用0x0000-0000~0xBFFF-FFFF这段虚拟地址空间。在Linux系统中,虽然不同的进程都使用相同的虚拟地址空间,但是这些虚拟地址空间相互独立,相互间不存在任何干扰。因为在Linux系统中,这些进程的虚拟地址空间被映射到不同的物理地址空间中。

基于以上这两种需求, Linux系统引入了虚拟内存技术,采用这种技术可以将无限大的虚拟地址空间映射到有效的物理地址空间中,同时有效管理Linux系统的核心空间。

在PowerPC处理器中,设置了专门的MMU和与此相关的寄存器,支持这种虚实地址转换。如在E500内核中设置了TLB1和TLB0两种硬件查找表,支持虚实地址转换。基于E500内核的Linux PowerPC,除了使用TLB1和TLB0进行虚实地址转换外,还使用了中断机制和一些数据结构,来完成虚实地址的转换。

(2) Linux PowerPC的核心空间的管理。如上文所示,对于32位PowerPC处理器,Linux PowerPC的核心空间在0xC000-000~0xFFFF-FFFF这段地址空间中。在这段空间中,包含两类物理内存:一类是主存储器,在32位处理器系统中,主存储器使用DDR或者SDRAM,主存储器主要用来存放Linux系统的程序与数据;另一类是外部设备,在PowerPC处理器中,外部设备使用的寄存器采用存储器映射方式进行寻址。

Linux PowerPC核心空间的管理主要针对主存储器。在Linux PowerPC中,主存储器的地址空间被映射到Linux PowerPC的核心空间中。对于32位PowerPC处理器,其可支持的主存储器的大小可能超过1 GB,而在Linux PowerPC中,核心空间最大也只有1 GB空间。对于这种主存储器容量过大的应用,Linux PowerPC需要使用HIMEM来管理高端内存,或者放弃对主存储器高端物理内存的使用。

目前基于32位处理器的Linux PowerPC,其1 GB大小的核心空间,已经显得捉襟见肘。这是因为在Linux PowerPC中,1 GB核心空间除了要映射主存储器空间之外,还要映射外部设备使用的地址空间,此外VM还要占用一部分核心空间。

在Linux PowerPC中,只能直接映射768 MB大小的主存储器空间。当Linux PowerPC使用大于768 MB的主存储器空间时,Linux PowerPC需要将这些高端内存空间(大于768MB的部分)映射到HIMEM中。有关HIMEM和VM的内容见下文。

Linux系统除了要将处理器系统中的物理地址空间映射到核心空间进行管理之外,还需要重点考虑如何高效地管理主存储器,如何尽可能地减少主存储器中的碎片。

Linux系统采用了许多种内存管理算法,如Buddy System和SLB分配器,优化Linux系统物理内存的访问效率,并尽量减少在物理内存中的碎片。Linux系统在引导时,还采用了一种特殊的物理内存——Boot Memory。在Linux系统进行引导时,只能使用Boot Memory中的物理内存,初始化Linux系统中其他子系统,包括内存管理子系统。

(3) 进程地址空间。在Linux系统中,每一个用户进程都有独立的用户地址空间,Linux系统需要对这些进程地址空间进行有效管理,使得这些进程地址空间不会互相干扰。有时进程地址空间还需要与文件系统进行交互,将可执行文件从硬盘等外部存储设备中读到主存储器中。

对于物理内存的管理及进程地址空间的管理这部分源代码来说,Linux PowerPC与基于其他体系的Linux系统,如Linux Pentium,并没有本质的不同。但是在进行虚实地址的转换时,Linux PowerPC与基于其他体系的Linux系统有本质的区别。

对于不同PowerPC处理器的内核,如603E内核和E500内核,其硬件MMU管理部件有较大的差异,因此不同内核的PowerPC处理器采用了不同的方法进行虚实地址转换。本书将以PowerPC E500内核为例对Linux PowerPC内存管理进行详细分析,同时兼顾基于603E内核的Linux PowerPC。

7.1 Linux PowerPC的虚实地址的转换

在Linux PowerPC中,虚实地址转换包含两部分内容。一是将处理器系统的物理地址空间映射到Linux核心空间中,如将物理地址为0000-0000～1FFFF-FFFF的DDR空间映射到虚拟地址为C000-0000～DFFF-FFFF的空间中。二是将用户进程的虚拟地址空间映射到主存储器所在的物理地址空间中,在Linux系统中,用户进程使用0000-0000～BFFF-FFFF之间的虚拟地址空间,这个虚拟地址空间必须要映射到物理地址空间中,才能被Linux系统直接访问。在Linux系统中,用户进程使用的3 GB大小的虚拟地址空间在同一时刻不会全部映射到物理地址空间中。

不同的PowerPC处理器提供了不同的MMU机制,支持虚实地址的转换。在本书的3.1节,我们学习了E500内核的TLB0和TLB1。其中,Linux PowerPC使用TLB1实现内存地址空间的段式映射,其大小可调。在E500 V1内核的TLB1中,支持的最大块为256 MB,最小为4KB。Linux PowerPC使用TLB0实现Linux PowerPC的页式映射,只能支持大小为4KB的物理页面。

在讲述Linux PowerPC的段页式地址映射之前,我们需要回顾E500内核硬件上提供的

一些虚实地址转换机制。如图 3-1 所示,E500 内核的虚拟地址一共由 41 位组成。而 Linux PowerPC 没有使用 E500 内核提供的 PID 寄存器和 AS 位,在 Linux PowerPC 中,并没有使用 E500 内核的地址空间 1,而只是使用 E500 内核的地址空间 0,在进行图 3-2 中的虚实地址转换时,将忽略 PID 寄存器。因此在 Linux PowerPC 中,虚拟地址等效于 E500 内核的有效地址。

在 Linux PowerPC 中,无论应用进程还是核心进程可以直接使用的地址只能是有效地址,这个有效地址为 32 位,由 EPN 和 offset 组成。与 603E 内核不同,E500 内核不能关闭 MMU,因此在 E500 内核中,用户或者核心进程不能直接访问物理地址而只有将有效地址通过 MMU 转换为物理地址后,才能访问相应的物理地址。

例如,Linux PowerPC 需要访问一段 256 MB 大小、物理内存区域为 0000-0000～0FFF-FFFF 的物理空间时,首先需要将这段物理内存区域映射到一段虚拟内存区域,如 0xC000-0000～0xCFFF-FFFF,之后通过访问这段虚拟地址对 0000-0000～0FFF-FFFF 这段物理内存区域进行操作。Linux PowerPC 可以使用段式映射或者页式映射进行虚实地址转换。

7.1.1 使用 TLB1 进行段式映射

在 E500 内核中, TLB1 一共有 16 个 Entry。Linux PowerPC 使用 TLB1 进行虚实地址的段式映射,采用段式映射可以直接使用硬件提供的 TLB1 查找表,而不使用在内存中的页表就可以完成虚实地址转换。这种虚实地址转换的效率较高。

但是在 E500 内核的 TLB1 中,Entry 数量有限,Linux PowerPC 不可能使用 TLB1 完成所有虚实地址转换。基于 E500 内核的 Linux PowerPC 可以使用 TLB1,对以下两类地址空间进行段式映射。

(1) Linux PowerPC 使用的核心空间。在基于 32 位处理器的 Linux PowerPC 中,Linux 内核占用最高端的 1 GB 空间,即 0xC000-0000～0xFFFF-FFFF,这段空间可以使用段式映射进行虚实地址转换。

(2) 外部设备使用的虚拟地址空间。这段地址空间需要映射到 Linux PowerPC 的 1 GB 核心空间中。Linux PowerPC 使用 ioremap 函数,将外部设备使用的物理地址映射为虚拟地址,之后 Linux PowerPC 通过这个虚拟地址对外部设备进行访问。

1. 使用段式映射的 Linux 核心空间

在 E500 内核中,Linux PowerPC 使用段式映射,将核心空间使用的物理内存进行虚实地址转换,之后 Linux 系统才能够使用这些虚拟地址空间。Linux PowerPC 的核心空间使用 0xC000-0000～0xFFFF-FFFF 之间的虚拟地址空间。

Linux PowerPC 使用 TLB1 的前 3 个 Entry 进行虚实地址转换,最大可以映射 768 MB 大小的物理地址空间,所以只能使用 0xC000-0000～0xEFFF-FFFF 之间的核心空间映射存放在主存储器中的物理地址空间。

对于一个主存储器大小为 768 MB 的处理器系统,如果其使用的物理地址为 0x0000-0000～0x3000-0000,那么 Linux PowerPC 将使用 0xC000-0000～0xEFFF-FFFF 这段虚拟地址空间与之映射。如果一个处理器系统的主存储器空间大于 768 MB,那么大于 768 MB 的这段高端地址空间将不能直接映射到 Linux 核心空间中,Linux 系统或者使用 HIMEM 映射这些大于 768 MB 的物理地址空间或者放弃对这些空间的访问。

在 Linux PowerPC 引导时，调用 MMU _ init→mapin _ ram→mmu _ mapin _ ram 函数将主存储器的物理地址空间映射到核心空间的 0xC000-0000～0xEFFF-FFFF 之间。mu _ mapin _ ram 函数在 ./arch/powerpc/mm/ fsl _ booke _ mmu.c 文件中，该函数源代码的详细说明如下：

```
unsigned long __ init mmu _ mapin _ ram(void)
{
  cam _ mapin _ ram(__ cam0, __ cam1, __ cam2);

  return __ cam0 + __ cam1 + __ cam2;
}
```

Linux PowerPC 使用 mmu _ mapin _ ram 函数建立 Linux 系统核心空间与物理地址空间的映射。该函数调用 cam _ maping _ ram 函数将__ cam0、__ cam1 和__ cam2 大小的虚拟地址空间映射到物理地址空间。__ cam0、__ cam1 和__ cam2 是 Linux PowerPC 定义的全局变量，这些变量的最大值为 cam _ max，其值为 0x1000-0000，即为 256 MB。

如果处理器系统一共有 768 MB 物理内存时，__ cam0，__ cam1 和__ cam2 的值相同都为 256 MB；如果一个系统中只有 260 MB 物理内存时，__ cam0 为 256 MB，__ cam1 为 4 MB 而__ cam2 为 0；如果一个系统中只有 64 MB 物理内存时__ cam0 为 64 MB，__ cam1 和__ cam2 都为 0。如果处理器系统的物理内存大于 768 MB 时，__ cam0，__ cam1 和__ cam2 的大小都为 256 MB，而此时剩余的物理内存不能使用 TLB1 进行段式映射，而必须使用 HIMEM 来管理。由此可见，基于 32 位 PowerPC 处理器的 Linux PowerPC，最多只能直接使用 768 MB 大小的主存储器。Linux PowerPC 使用 adjust _ total _ lowmem 函数计算__ cam0，__ cam1 和__ cam2 变量，该函数在系统引导时由 MMU _ init 函数调用，其源代码详解如下：

```
void __ init
adjust _ total _ lowmem(void)
{
  unsigned long max _ low _ mem = MAX _ LOW _ MEM;
  unsigned long cam _ max = 0x10000000;
  unsigned long ram;
  /* adjust CAM size to max _ low _ mem */
  if (max _ low _ mem < cam _ max)
    cam _ max = max _ low _ mem;
  /* adjust lowmem size to max _ low _ mem */
  if (max _ low _ mem < total _ lowmem)
    ram = max _ low _ mem;
  else
    ram = total _ lowmem;
```

adjust _ total _ lowmem 函数的执行流程如下：

- 首先对变量 max _ low _ mem 和 cam _ max 赋值。max _ low _ mem 变量存放 Linux PowerPC 低端内存大小，其值为 MAX _ LOW _ MEM，该值在配置 Linux PowerPC 时初始化，MAX _ LOW _ MEM 等效于 CONFIG _ LOWMEM _ SIZE。在 Linux PowerPC

中，其值为 768 MB。而 cam _ max 参数存放 TLB1 的一个 Entry 可以映射物理内存的最大值，对于 E500 V1 内核，其值为 256 MB。

● 这段程序根据 max _ low _ mem 和 total _ lowmem 的值调整 cam _ max 和 ram 参数的大小。全局变量 total _ lowmem 存放当前处理器系统可以访问的内存大小，而 ram 参数存放使用 TLB1 进行虚实地址转换的内存大小。

```
/* Calculate CAM values */
__cam0 = 1UL << 2 * (__ilog2(ram) / 2);
if (__cam0 > cam_max)
  __cam0 = cam_max;
ram -= __cam0;
if (ram) {
  __cam1 = 1UL << 2 * (__ilog2(ram) / 2);
  if (__cam1 > cam_max)
     __cam1 = cam_max;
  ram -= __cam1;
}
if (ram) {
  __cam2 = 1UL << 2 * (__ilog2(ram) / 2);
  if (__cam2 > cam_max)
    __cam2 = cam_max;
  ram -= __cam2;
}
printk(KERN_INFO"Memory CAM mapping: CAM0 = %ldMb, CAM1 = %ldMb,"
    " CAM2 = %ldMb residual: %ldMb\n",
    __cam0 >> 20, __cam1 >> 20, __cam2 >> 20,
    (total_lowmem - __cam0 - __cam1 - __cam2) >> 20);
__max_low_memory = max_low_mem = __cam0 + __cam1 + __cam2;
}
```

这段源代码首先确定全局变量__ cam0，__ cam1 和__ cam2 的大小，之后根据这些全局变量的值，调整全局变量__ max _ low _ memory 和 max _ low _ mem 的值。

全局变量__ cam0，__ cam1 和__ cam2 的大小确定后，调用 cam _ mapin _ ram 函数进行段式映射。cam _ mapin _ ram 函数的源代码在 ./arch/powerpc/mm/fsl _ booke _ mmu.c 文件中。其源代码的详解如下：

```
void __init cam_mapin_ram(unsigned long cam0, unsigned long cam1,
    unsigned long cam2)
{       ⋮
```

cam _ mapin _ ram 函数一共有 3 个输入参数，分别为 cam0，cam1 和 cam2。这 3 个参数分别与__ cam0，__ cam1 和__ cam2 变量对应，该函数没有返回值。在该函数中，最多可以使用 E500 内核 TLB1 中的 3 个 Entry，分别是 Entry 0，1 和 2，将 Linux PowerPC 的核心空间与物理地址空间进行段式映射。

```
    settlbcam(0, KERNELBASE, PPC_MEMSTART,
              cam0, _PAGE_KERNEL, 0);
    tlbcam_index++;
    if (cam1) {
    tlbcam_index++;
    settlbcam(1, KERNELBASE + cam0, PPC_MEMSTART + cam0,
              cam1, _PAGE_KERNEL, 0);
    }
    if (cam2) {
    tlbcam_index++;
    settlbcam(2, KERNELBASE + cam0 + cam1, PPC_MEMSTART + cam0 + cam1,
              cam2, _PAGE_KERNEL, 0);
    }
} /* End cam_mapin_ram */
```

在 cam_mapin_ram 函数中,宏 KERNELBASE(其值为 0xC000-0000)描述 Linux 内核虚拟地址的基地址,而宏 PPC_MEMSTART(其值为 0x0000-0000)描述 Linux 内核物理地址的基地址,因此在这段程序中,最多可以将 Linux 系统 0xC000-0000～0xEFFF-FFFF 之间的虚拟地址空间,映射到 0x0000-0000～0x2FFF-FFFF 之间的物理地址空间中。

Linux PowerPC 假定处理器系统中使用的物理地址空间的基地址为 0,如果用户使用的物理地址空间的基地址不为 0,则需要对宏 PPC_MEMSTART 进行调整。

cam_mapin_ram 函数调用 settlbcam 函数,操作 MAS 寄存器对 TLB1 的 Entry 0,1 和 2 进行设置。settlbcam 函数在 ./arch/powerpc/mm/fsl_booke_mmu.c 文件中,其源代码详解如下:

```
void settlbcam(int index, unsigned long virt, phys_addr_t phys,
        unsigned int size, int flags, unsigned int pid)
{
    unsigned int tsize, lz;
```

settlbcam 函数一共有 6 个参数。

(1) index 参数。该参数确定 settlbcam 函数使用 TLB1 中的哪一个 Entry。

(2) virt 参数存放用来建立段映射关系的虚拟地址。

(3) phys 参数存放用来建立段映射关系的物理地址。

(4) size 参数用来存放段的大小,其取值范围与具体的硬件有关,在 E500 V1 内核中,size 参数的取值范围为 4 KB～256 MB。

(5) flags 参数用来设置 TLB1 相应 Entry 的属性。其可能的值有_PAGE_KERNEL,_PAGE_KERNEL_RO,_PAGE_IO,_PAGE_RAM 等,这些属性主要描述 Entry 中的 WIMG 和 PERMIS 参数。

其中较为常用的属性有_PAGE_IO,_PAGE_RAM。_PAGE_IO 属性对外部设备进行描述,Linux PowerPC 为外部设备空间建立虚实映射,将使用_PAGE_IO 属性;_PAGE_RAM 对主存储器进行描述,Linux PowerPC 为主存储器空间建立虚实映射,将使用_PAGE_

RAM 属性。

(6) pid 参数与 MAS1 寄存器中的 TID 字段对应。

```
    asm ("cntlzw %0,%1" : "=r" (lz) : "r" (size));
    tsize = (21 - lz) / 2;

#ifdef CONFIG_SMP
    if ((flags & _PAGE_NO_CACHE) == 0)
      flags |= _PAGE_COHERENT;
#endif
```

这段程序首先将输入参数中的 size 参数,转换为 MAS1 寄存器使用的 SIZE 字段。如果 Linux PowerPC 支持 SMP 结构,且 flags 参数中 WIMGE 字段的 I 位没有使能,则将 M 位置为 1。WIMGE 字段的说明见 3.3.3 节。

```
    TLBCAM[index].MAS0 = MAS0_TLBSEL(1) | MAS0_ESEL(index) |
                         MAS0_NV(index+1);
    TLBCAM[index].MAS1 = MAS1_VALID | MAS1_IPROT | MAS1_TSIZE(tsize) |
                         MAS1_TID(pid);

    TLBCAM[index].MAS2 = virt & PAGE_MASK;

    TLBCAM[index].MAS2 |= (flags & _PAGE_WRITETHRU) ? MAS2_W : 0;
    TLBCAM[index].MAS2 |= (flags & _PAGE_NO_CACHE) ? MAS2_I : 0;
    TLBCAM[index].MAS2 |= (flags & _PAGE_COHERENT) ? MAS2_M : 0;
    TLBCAM[index].MAS2 |= (flags & _PAGE_GUARDED) ? MAS2_G : 0;
    TLBCAM[index].MAS2 |= (flags & _PAGE_ENDIAN) ? MAS2_E : 0;

    TLBCAM[index].MAS3 = (phys & PAGE_MASK) | MAS3_SX | MAS3_SR;
    TLBCAM[index].MAS3 |= ((flags & _PAGE_RW) ? MAS3_SW : 0);

#ifndef CONFIG_KGDB /* want user access for breakpoints */
    if (flags & _PAGE_USER) {
      TLBCAM[index].MAS3 |= MAS3_UX | MAS3_UR;
      TLBCAM[index].MAS3 |= ((flags & _PAGE_RW) ? MAS3_UW : 0);
    }
#else
...
#endif
```

这段程序将根据 flags、index、virt 和 phys 的值初始化 E500 内核的 MAS0、MAS1、MAS2 和 MAS3 寄存器。

```
    tlbcam_addrs[index].start = virt;
    tlbcam_addrs[index].limit = virt + size - 1;
    tlbcam_addrs[index].phys = phys;
```

```
        loadcam_entry(index);
    }
```

这段程序的执行流程如下：

- 更新 tlbcam_addrs 数组，在 tlbcam_addrs 数组中存放着 Linux PowerPC 段式映射的虚实地址转换关系。
- 使用 loadcam_entry 函数，将存放在 TLBCAM[index]中的 MAS0，MAS1，MAS2 和 MAS3 寄存器写入对应的 TLB1 的 entry 中。

loadcam_entry 函数的源代码在 ./arch/powerpc/kernel/head_fsl_booke.S 文件中。

2. 使用段式映射的外部设备地址空间

在 Linux 系统中，设备驱动程序使用 ioremap 函数，将外部设备的物理地址空间转换为虚拟地址空间，然后使用虚拟地址空间对外部设备进行访问。ioremap 函数使用段式映射或者页式映射得到相应的虚拟地址空间。如果 ioremap 函数希望使用段式映射获得虚拟地址空间，Linux PowerPC 必须事先使用 io_block_mapping 函数，建立虚拟地址与物理地址的段式映射。

在设备驱动程序中，就是否采用虚实地址的段式映射引发了许多争执。Linux PowerPC 的开发团队最终决定在设备驱动程序中放弃段式映射，即不再使用 io_block_mapping 函数为设备驱动程序建立虚拟地址与物理地址之间的段式映射。当设备驱动程序使用 ioremap 函数时，只能通过页式映射获得相应设备的虚拟地址。

Linux PowerPC 的开发团队取消 io_block_mapping 函数的主要理由是，Linux PowerPC 内核的维护者发现，有许多用户滥用 io_block_mapping 函数，从而使得一个完整的 Linux 内核空间被分割得支离破碎，这对 Linux PowerPC 的维护十分不利。

而且有些系统程序员并没有充分理解 Linux 系统内存子系统，将外部设备的物理地址空间错误地映射到了 Linux 核心空间中。在 Linux PowerPC 中，外部设备可以使用的 Linux 核心空间是指定的，用户不能对此任意选择。

目前，Linux PowerPC 的维护者使用页式内存映射管理外部设备的物理地址空间。但是，如果用户需要编写较为高速的设备驱动程序，还可以使用段式映射管理外部设备使用的物理地址空间，为此 Linux PowerPC 依然保留了 io_block_mapping 函数。

io_block_mapping 函数在 ./arch/powerpc/mm/pgtable_32.c 文件中，这个函数的主要作用是，将外部设备使用的物理地址空间与 Linux PowerPC 核心空间建立段式映射。在 E500 内核中，TLB1 一共有 16 个 Entry，其中 TLB1 的 Entry 0，1 和 2 已经被 Linux 核心空间的段式映射使用，因此 Linux PowerPC 可以使用剩余的 13 个 Entry，即第 3～15 个 Entry 用来建立虚实地址映射。当这些 Entry 全部使用完毕后，io_block_mapping 函数需要使用页式管理方式建立虚实地址的映射。io_block_mapping 函数的源代码详解如下：

```
/*
 * Set up a mapping for a block of I/O.
 * virt, phys, size must all be page-aligned.
 * This should only be called before ioremap is called.
 */
```

```
void __init io_block_mapping(unsigned long virt, phys_addr_t phys,
            unsigned int size, int flags)
{
  int i;
```

io_block_mapping函数共有4个参数,分别为virt、phys、size和flags。这些参数与settlbcam函数的相应参数一致。在这段源代码的注释中可以发现,该函数需要在设备驱动程序使用ioremap函数之前被调用。

有许多用户混淆了io_block_mapping函数和ioremap函数的用法,有些用户甚至认为这两个函数并没有什么本质区别。实际上这两个函数没有任何联系,io_block_mapping函数用来建立外部设备虚实地址之间的段式映射,而ioremap函数是从已经建立了虚实地址映射的空间中获得虚拟地址。在7.3.5节中将详细介绍ioremap函数。

```
if (virt > KERNELBASE && virt < ioremap_bot)
  ioremap_bot = ioremap_base = virt;
```

io_block_mapping函数首先对virt参数进行参数检查。程序员使用io_block_mapping函数建立虚实地址的映射时,需要注意,io_block_mapping函数使用的虚拟地址必须大于KERNELBASE并且小于ioremap_bot。在ioremap_bot和ioremap_base变量中,记载着设备驱动程序可以使用的虚存的基地址,这两个变量的详细说明见下文。当virt参数在这段地址空间时,将调整ioremap_bot和ioremap_base变量的值。

```
#ifdef HAVE_BATS
  /*
   * Use a BAT for this if possible...
   */
  if (io_bat_index < 2 && is_power_of_2(size)
    && (virt & (size - 1)) == 0 && (phys & (size - 1)) == 0) {
    setbat(io_bat_index, virt, phys, size, flags);
    ++io_bat_index;
    return;
  }
#endif /* HAVE_BATS */
```

这段程序对基于603E和604E的PowerPC处理器有效,在这些PowerPC处理器中不支持TLB1,而是使用IBAT或者DBAT。该类处理器使用setbat函数建立虚实地址的段式映射。603E内核和604E内核使用哈佛结构将指令BAT和数据BAT分离出来,而在E500内核中,L1 MMU使用哈佛结构将L1指令TLB和L1数据TLB进行分离。

```
#ifdef HAVE_TLBCAM
  /*
   * Use a CAM for this if possible...
   */
  if (tlbcam_index < num_tlbcam_entries && is_power_of_4(size)
```

```
        && (virt & (size - 1)) == 0 && (phys & (size - 1)) == 0) {
        settlbcam(tlbcam_index, virt, phys, size, flags, 0);
        ++tlbcam_index;
        return;
    }
#endif /* HAVE_TLBCAM */
```

基于 E500 内核的 Linux Power,HAVE_TLBCAM 为 1,而 HAVE_BATS 为 0。在这段函数中,首先检查 tlbcam_index 参数是否超过系统的最大值,之后对 size、virt 和 phys 参数进行对界检查。E500 内核要求 tlbcam_index 参数不能大于 16,size 参数为 4 字节对界,而 virt 和 phys 参数以 size 参数所要求的大小进行对界。当参数检查通过后,调用 settlbcam 函数对 E500 内核 TLB1 的相应 Entry 进行设置以完成虚实地址的映射,然后返回。

```
    /* No BATs available, put it in the page tables. */
    for (i = 0; i < size; i += PAGE_SIZE)
        map_page(virt + i, phys + i, flags);
} /* End io_block_mapping */
```

如果在 TLB1 中没有可用的 Entry,Linux PowerPC 使用 map_page 函数建立虚拟地址到物理地址的页式映射。

7.1.2 使用 TLB0 进行页式映射

在 Linux 系统中,绝大多数虚拟地址空间使用页式方式映射。

- 用户进程的虚拟地址空间使用 0x0000-0000～0xBFFF-FFFF 之间的数据区域,这段虚拟地址空间使用页式映射方式。值得注意的是,在 Linux 系统中,每一个用户进程都有属于自己的 3 GB 空间,这些空间彼此独立。因此在 Linux 系统中,用户进程使用虚拟地址空间的大小为 3 GB×进程数。
- 外部设备使用的虚拟地址空间也使用页式方式进行映射。在 7.1.1 节中,我们讲述过外部设备的段式映射方式。但是在绝大多数情况下,Linux PowerPC 使用页式映射方式管理外部设备使用的虚拟地址空间。

基于 E500 内核的 Linux PowerPC,使用 TLB0 和系统页表完成虚实地址的转换,这种虚实地址转换也被称为页式映射。其中 TLB0 存放在 E500 内核中,而系统页表存放在主存储器中。在 E500 V1 内核中,TLB0 中一共有 256 个 Entry,其中每个 Entry 只能映射 4KB 大小的页面。因此在 TLB0 中最多只能保存 256×4 KB=1 MB 大小的虚拟地址空间。

对于 32 位 PowerPC 处理器,TLB0 能直接映射的虚拟地址空间远远不够。因此处理器访问页式映射的虚拟地址空间时,将不可避免地出现 TLB Miss 异常。所谓 TLB Miss 异常是指处理器访问的虚拟地址空间不在 TLB0 的 Entry 中。E500 内核对数据空间访问产生的 TLB Miss 被称为数据 TLB Miss,简称为 DTLB Miss;对程序空间访问产生的 TLB Miss 被称为程序 TLB Miss,简称为 ITLB Miss。

在 E500 内核中,ITLB Miss 或者 DTLB Miss 出现时,将引发 Instruction TLB Error Interrupt 异常或者 Data TLB Error Interrupt 异常。在这些异常处理程序中,Linux PowerPC 查找

系统页表，获得与被访问虚拟地址对应的物理地址，并同步 TLB0。

读者可以这样简单理解 TLB0：在 Linux PowerPC 中，TLB0 作为系统页表的 Cache，使用页式方式进行虚实地址映射的虚拟地址，当发生 TLB0 Miss 时，首先在 TLB0 这个"Cache"中，查找与虚拟地址对应的物理地址，如果在 TLB0 中没有找到这个映射关系，则在系统页表中继续查找这个映射关系，之后同步页表与这个 Cache。

1. Linux PowerPC 的系统页表

Linux PowerPC 进入 Instruction TLB Error Interrupt 异常或者 Data TLB Error Interrupt 异常处理程序时，将根据虚拟地址查找系统页表。在 Linux 系统中，所有使用页式映射的物理内存被分为一个个固定大小的空间进行管理，对于 32 位处理器，这个空间的大小为 4KB。Linux PowerPC 使用 pte_t 结构，描述这个 4KB 大小的虚拟地址空间与物理地址空间之间的映射关系。

为了实现这种映射关系，Linux 系统将虚拟地址分解为 PGD(Page Global Directory)，PUD (Page Upper Directory)、PMD(Page Middle Directory)、PTE(Page Table Entry)和 Offset 共 5 个字段。Linux 系统通过虚拟地址的 PGD、PMD 和 PTE 字段计算出该虚拟地址使用的物理页面 pte_t，然后从这个 pte_t 结构中获得该虚拟地址使用的物理地址。

在 Linux 系统中，使用了一些宏描述虚拟地址 PGD、PUD、PMD 和 PTE 字段的一些属性，这些宏的定义在 ./include/asm-powerpc/pgtable.h 文件中。在 Linux 2.6.20 内核中，没有完全移除 ./include/asm-ppc 目录下的文件，Linux PowerPC 还需要使用该目录中的某些文件。

在 ./include/asm-powerpc/pgtable.h 文件中，需要对 CONFIG_PPC64 条件进行判断，如果该条件为假，即当前 Linux PowerPC 是针对 32 位 PowerPC 处理器时，将引用 ./include/asm-ppc/pgtable.h 文件，其代码如下：

```
#ifndef _ASM_POWERPC_PGTABLE_H
#define _ASM_POWERPC_PGTABLE_H
#ifdef __KERNEL__

#ifndef CONFIG_PPC64
#include <asm-ppc/pgtable.h>
```

Linux PowerPC 与 PGD 和 PMD 定义有关的宏在 ./include/asm-ppc/pgtable.h 文件中，与 PTE 和 PAGE 有关的宏分别在 ./include/asm-powerpc/page_32.h 和 ./include/asm-powerpc/page.h 文件中。这些宏如表 7-1 所示。

表 7-1 与系统页表有关的宏

名　称	值	名　称	值
PAGE_SHIFT	12	PMD_SIZE	0x0040-0000
PAGE_MASK	0xFFFF-F000	PMD_MASK	0xFFC0-0000
PAGE_SIZE	0x0000-1000	PGDIR_SHIFT	22
PTE_SHIFT	10	PGDIR_SIZE	0x0040-0000
PMD_SHIFT	22	PGDIR_MASK	0xFFC0-0000

这些宏与 Linux PowerPC 虚拟地址之间的关系如图 7-1 所示。

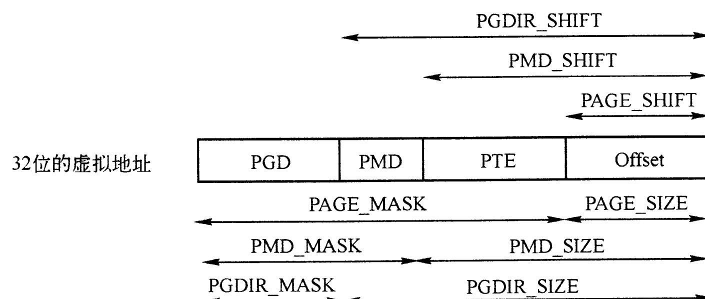


图 7-1　虚拟地址与 PGD，PMD，PTE 相关宏的关系

如上图所示，在 32 位 PowerPC 处理器中，Linux PowerPC 虚拟地址的 PGD 字段的长度为 32-PGDIR _ SHIFT＝10；PMD 字段的长度为 PGDIR _ SHIFT- PMD _ SHIFT＝0，由此可见，对于 32 位 PowerPC 处理器，Linux PowerPC 的虚拟地址不支持 PMD 字段；PTE 字段的长度为 PMD _ SHIFT- PAGE _ SHIFT＝10；Offset 字段的长度为 PAGE _ SHIFT＝12。

Linux PowerPC 将 32 虚拟地址分解为 10 位的 PGD，10 位的 PTE 和 12 位的 Offset 的主要目的是减少空间的浪费。在 Linux 系统中，一个普通进程可以使用 3 GB 虚拟地址连续的空间，但是绝大多数进程并不需要这么大的虚拟内存。因此在虚拟地址连续的 3 GB 进程空间中不可避免地会有许多空洞，为此 Linux 系统采用二级映射方式使用虚拟地址到物理地址的转换以避开这些空洞，提高内存的利用率。

在 Linux 系统中，每一个进程都有自己的 PGD 表，PGD 表的每一个 Entry 与 4 MB 虚拟内存相对应。如果与 PGD 表 Entry 相对应的虚拟地址内存没有被使用，则将 PGD 表中的 Entry 置为空，从而可以节省一部分空间；否则 PGD 表的 Entry 指向一个 PTE 表。PTE 表的每一个 Entry 由两部分组成：一是与虚拟地址页面（Virtual Page Number，VPN）对应的物理地址页面（Real Page Number，RPN）；二是对此物理页面进行描述的属性。

在 Linux 系统中，每一个用户进程都有自己的 PGD 基地址，存放在相应进程描述符的 mm－>pgd 中；由 5.1.1 节所示，Linux 核心进程的 mm 参数为空，因此核心进程的 PGD 基地址为 NULL。Linux 系统规定，核心进程使用 swapper _ pg _ dir 作为 PGD 基地址。

在 Linux 系统中，虚拟内存地址采用下列步骤进行页式映射。本书再次提醒读者注意，如果一个虚拟地址在 TLB0 中命中，则从 TLB0 中直接获得其需要的物理地址，而不用使用下列步骤。下列步骤在 Linux PowerPC 出现 ITLB Miss 或者 DTLB Miss 时使用：

(1) Linux PowerPC 首先需要通过 mm→pgd 或者 swapper _ pg _ dir 和当前虚拟地址的 PGD 字段获得与该地址对应的 pgd _ t 结构。

(2) 然后通过 pgd _ t 结构与虚拟地址的 PTE 字段获得与该地址对应的 pte _ t 结构。

(3) 在 Linux PowerPC 中，每一个物理页面使用一个 pte _ t 结构。在 pte _ t 结构中，存放与虚拟地址相对应的物理页表基址指针和对该物理页表的描述。

(4) 使用 pte _ t 表的 VPN 字段和虚拟地址的 Offset，通过计算，获得与虚拟地址相对应的物理地址。

以上步骤的示意如图 7-2 所示。

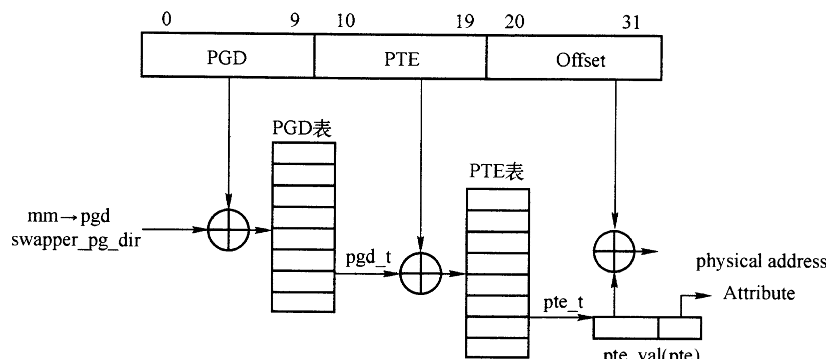


图 7-2 页式地址映射示意图

在基于 E500 内核的 Linux PowerPC 中，使用长整型 long 描述 pte _ t 结构，pte _ t 结构由两部分组成，物理页面的基地址和该物理页面的属性，其中物理页面的基地址占用高 20 位，而物理页面的属性占用低 12 位，Linux PowerPC 的 pte _ t 结构如下所示：

0～19	20	21	22	23	24	25	26	27	28	29	30	31
RPN	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0

pte _ t 结构中每一个字段或者位的含义如下所示：

- RPN 字段占用 pte _ t 的第 0～19 位，描述物理页面的基地址。
- B11，_ PAGE _ DIRTY 位。为 1 时表示当前物理页面刚刚被改写。
- B10，_ PAGE _ WRITETHRU 位。该位与 3.3.3.1 节中 WIMG 字段的 W 位对应。
- B9，_ PAGE _ NO _ CACHE 位。该位与 3.3.3.1 节中 WIMG 字段的 I 位对应。
- B8，_ PAGE _ COHERENT 位。该位与 3.3.3.1 节中 WIMG 字段的 M 位对应。
- B7，_ PAGE _ GUARDED 位。该位与 3.3.3.1 节中 WIMG 字段的 G 位对应。
- B6，_ PAGE _ ENDIAN。该位与 3.3.3.1 节中 WIMG 字段的 E 位对应。
- B5，_ PAGE _ HWEXEC。该位为 1 时表示该页面可以用作程序空间，该位与 3.1.2 节中的 SX 和 UX 位相关。
- B4，_ PAGE _ RW。该位为 1 时表示该页面可读写，该位是一个软件位，而 B3 位与硬件直接相关。
- B3，_ PAGE _ HWWRITE。该位为 1 时表示该页面可读写，该位与 3.1.2 节中的 UW，UR，SW 和 SR 位相关。
- B2，_ PAGE _ ACCESSED。该位为 1 时表示该页面正在使用。
- B1，_ PAGE _ USER。该位为 1 时表示该页面被在用户空间的进程使用。
- B0，_ PAGE _ PRESENT。该位为 1 时表示该页面在物理内存而不是在对换区中。

在 Linux 系统中，还有一个 page 结构也用来描述这个物理页面。但是 pte _ t 结构和 page 结构的含义完全不同，pte _ t 结构实现虚拟地址的页式映射，其中存放着与虚拟地址对应的物理地址和与 MMU 有关的属性描述；而 page 结构存放着物理页面的描述信息，即便是采用段式映射的虚拟地址空间也有一个 page 结构，page 结构被称为页表描述符。本书将在下文详细介绍 page 结构。

2. Linux PowerPC 系统页表的创建

由上文可知,Linux PowerPC 的系统页表由三部分组成,分别为 PGD 表、PGD 表和 PTE 表。在 Linux 系统中,核心进程和所有普通进程都有自己的一套系统页表。

Linux 系统核心进程使用 swapper_pg_dir 作为 PGD 的基地址,该值的定义如下所示:

```
    .data
    .align  12
    .globl  sdata
sdata:
    .globl  empty_zero_page
empty_zero_page:
    .space  4096
    .globl  swapper_pg_dir
swapper_pg_dir:
    .space  4096
```

由上述程序得知,swapper_pg_dir 的大小为 4096B,可以描述 1024 个 PGD 表 Entry。当 Linux 系统被加载后,该段区域所使用的物理内存被静态分配。

应用程序的 PGD 表基地址在进程描述符的 mm->pgd 参数中保存,该参数在进程创建时,由 copy_mm->dup_mm->mm_init->mm_alloc_pgd 函数初始化。

mm_alloc_pgd 函数调用 pgd_alloc 函数,为当前进程的 PGD 表分配空间,并将 PGD 表的基地址赋给 mm->pgd。pgd_alloc 函数在 ./arch/powerpc/mm/pgtable_32.c 文件中,该函数调用__get_free_pages 从 Linux 系统的 Buddy System 中获得当前进程 PGD 表所需要的空间,其源代码如下所示:

```
pgd_t *pgd_alloc(struct mm_struct *mm)
{
  pgd_t *ret;
  ret = (pgd_t *)__get_free_pages(GFP_KERNEL|__GFP_ZERO, PGDIR_ORDER);
  return ret;
}
```

Linux 系统在创建进程时,仅需要申请 PGD 表所需的物理空间,并不会初始化 PGD 表中的 Entry,也不会建立相应的 PTE 表。

这种做法主要基于两种考虑:一是为了加快进程创建的速度;二是在进程创建时无法确定需要初始化哪些 PGD 和 PTE 表。Linux 系统采用了 On-Demand 策略,即"用时创建"的策略,建立进程的 PGD 和 PTE 表。

在 Linux 系统中,如果进程使用的虚拟地址没有在 TLB0 中命中,将启动 Instruction TLB Error Interrupt 异常或者 Data TLB Error Interrupt 异常处理程序,在 PTE 表中搜索合适的 Entry,如果在 TLB Error Interrupt 异常程序中,仍然没有获得合适的 Entry,则创建 PTE 表的相应 Entry,或者从 Linux 系统的 SWAP 区中获得合适的 Entry。

当用户程序第一次访问使用页式映射的虚拟地址时,会进入 Data Storage Interrupt 异常或者 Instruction Storage Interrupt 异常处理程序创建 PTE 表的相应 Entry,因为此时在 TLB0

中不会有该页面对应的信息。Linux PowerPC 将会在 Data Storage Interrupt 异常或者 Instruction Storage Interrupt 异常处理程序调用__handle_mm_fault 函数，该函数将建立 PTE 表相应的 Entry，其源代码如下所示：

```
int __handle_mm_fault(struct mm_struct *mm, struct vm_area_struct *vma,
    unsigned long address, int write_access)
{
  pgd_t *pgd;
  pud_t *pud;
  pmd_t *pmd;
  pte_t *pte;

  __set_current_state(TASK_RUNNING);
  count_vm_event(PGFAULT);

  if (unlikely(is_vm_hugetlb_page(vma)))
    return hugetlb_fault(mm, vma, address, write_access);
```

__handle_mm_fault 函数首先将当前进程状态设置为 TASK_RUNNING，然后进行一些参数检查。

```
  pgd = pgd_offset(mm, address);
  pud = pud_alloc(mm, pgd, address);
  if (!pud)
    return VM_FAULT_OOM;
  pmd = pmd_alloc(mm, pud, address);
  if (!pmd)
    return VM_FAULT_OOM;
  pte = pte_alloc_map(mm, pmd, address);
  if (!pte)
    return VM_FAULT_OOM;

  return handle_pte_fault(mm, vma, address, pte, pmd, write_access);
}
```

这段程序首先使用 pgd_offset 函数，确定当前地址 address 使用的 pgd 表项，之后使用 pte_alloc_map 函数，在 Buddy System 中分配一个 PTE 表，其大小为 4KB。在基于 32 位 PowerPC 处理器的 Linux PowerPC 中，PUD 和 PMD 表没有被使用，此时 pud_alloc 函数的返回值为 pgd，而 pmd_alloc 函数的返回值为 pud。在这段函数中 pgd，pud 和 pmd 三个参数的值相同。最后该程序调用 handle_pte_fault 函数处理相应的 TLB Miss，该函数的详细描述见本书的 7.5.2 节。

3. Data TLB Error Interrupt 异常处理

E500 内核对使用页式映射的虚拟地址进行访问时，如果这个虚拟地址没有在 TLB0 中命中，E500 内核会产生 TLB Error Interrupt 异常。如果这个虚拟地址属于程序空间则产生 In-

struction TLB Error Interrupt 异常；如果这个虚拟地址属于数据空间则产生 Data TLB Error Interrupt 异常。

在 Linux 系统中，Instruction TLB Error Interrupt 异常和 Data TLB Error Interrupt 异常的处理较为相似，本书只介绍 Data TLB Error Interrupt 异常。在 Linux PowerPC 进入 Data TLB Error Interrupt 异常处理程序之前，E500 内核将表 7-1 所示的寄存器由硬件进行保存，以加快处理速度。

表 7-2　E500 内核进入 TLB Miss 异常后自动保存的寄存器

寄存器名称	更新后的值	
SRR0	保存被中断指令的有效地址，以便异常处理结束后返回	
SRR1	保存发生中断时的 MSR 寄存器的值，以便异常处理结束后恢复现场	
MSR	CE，ME 和 DE 位不变，其他位清零	
DEAR	保存引发 Data TLB Error Interrupt 异常的数据有效地址	
ESR	当存储器写操作，dcbi 或者 dcbz 指令引发 Data TLB Error Interrupt 异常时，该寄存器的 ST 位将置为 1 并将该寄存器的其他位清零	
MSRn	TLBSEL	赋值为保存在 MSR4 寄存器中 TLBSELD 字段
	ESEL	如果 TLBSEL 为 0，即选择 TLB0 时将 ESEL 字段赋值为 TLB0[NV]
	NV	如果 TLBSEL 为 0，将 NV 赋值为！TLB0[NV]
	IPROT	0
	TID[0-7]	赋值为保存在 MSR4 寄存器中 TIDSELD 字段
	TS	赋值为当前 MSR 寄存器的 DS 位
	TSIZE[0-3]	赋值为保存在 MSR4 寄存器中 TSIZED 字段
	EPN[32:51]	存放引发 Data TLB Error Interrupt 异常的数据的有效地址 EPN
	X0，X1，W，I，M，G，E	赋值为保存在存放在 MSR4 寄存器中的 X0D，X1D，WD，ID，MD，GD，ED
	RPN[32-51]	该中断处理结束时将从系统页表中获得对应的 RPN
	PERMIS	0
	SPID	PID0
	SAS	MSR 寄存器的 DS 位

由上表所示，Linux PowerPC 进行 Data TLB Error Interrupt 异常处理之前，E500 内核已经将 MSRn 中的许多寄存器由硬件完成赋值，从而有效提高了 Data TLB Error Interrupt 异常处理的效率。基于 E500 内核的 Linux PowerPC 通过以下语句，将 Data TLB Error Interrupt 异常处理程序的入口地址设置到 IVPR 和 IVOR13 的寄存器中。

```
SET_IVOR(13,DataTLBError);
```

当 Data TLB Error Interrupt 异常发生时，Data TLB Error Interrupt 异常处理程序将在 IVPR 和 IVOR13 寄存器指定的地址处执行。在 Linux PowerPC 中，Data TLB Error Interrupt 异常处理程序为 DataTLBError 函数。

DataTLBError 函数的定义在 ./arch/powerpc/kernel/head_fsl_booke.S 文件中。当外部

中断事件发生时，Linux PowerPC 将从 DataTLBError 函数开始执行，对 Data TLB Error Interrupt 异常进行处理。DataTLBError 函数的主要代码如下：

```
/* Data TLB Error Interrupt 源代码片段 1 */
    START_EXCEPTION(DataTLBError)
    mtspr SPRN_SPRG0, r10          /* Save some working registers */
    mtspr SPRN_SPRG1, r11
    mtspr SPRN_SPRG4W, r12
    mtspr SPRN_SPRG5W, r13
    mfcr r11
    mtspr SPRN_SPRG7W, r11
    mfspr r10, SPRN_DEAR           /* Get faulting address */
```

这段程序的执行流程如下：

- 程序首先将寄存器 r10, r11, r12 和 r13 分别保存在 SPRG0, SPRG1, SPRG4W 和 SPRG5W 寄存器中。注意，在 DataTLBError 函数中，不能使用堆栈保留任何寄存器和临时变量。
- 将 CR 寄存器的值存入 SPRG7W 寄存器中。
- 将 DEAR 寄存器中的数值赋值到 r10 寄存器中，从而获得引发 Data TLB Error Interrupt 异常的数据有效地址。

```
/* Data TLB Error Interrupt 源代码片段 2 */
    lis      r11, TASK_SIZE@h
    ori      r11, r11, TASK_SIZE@l
    cmplw    5, r10, r11
    blt      5, 3f
    lis      r11, swapper_pg_dir@h
    ori      r11, r11, swapper_pg_dir@l

    mfspr r12,SPRN_MAS1         /* Set TID to 0 */
    rlwinm   r12,r12,0,16,1
    mtspr    SPRN_MAS1,r12

    b  4f

    /* Get the PGD for the current thread */
3:
  mfspr      r11,SPRN_SPRG3
  lwz        r11,PGDIR(r11)
```

这段程序的执行流程如下：

- 比较 DEAR 寄存器和 TASK_SIZE 参数。并根据比较结果，判断引发当前 Data TLB Error Interrupt 异常的数据（该数据保存在 DEAR 寄存器中）是来自 Linux 内核空间还是用户空间。TASK_SIZE 的值为 0xC0000000，如果 DEAR 寄存器的值小于 TASK_SIZE，则表示 Linux 系统在访问用户空间时引发了 Data TLB Error Interrupt 异常，否则

表示 Linux 系统在访问核心空间时引发了 Data TLB Error Interrupt 异常。

- 获得当前进程使用的 PGD 表基地址,并保存到 r11 寄存器中。如果对核心空间的数据进行访问时,引发了 Data TLB Error Interrupt 异常,则将 swapper _ pg _ dir 赋值到 r11 寄存器中,否则将当前进程描述符的 thread.pgdir 参数赋值到 r11 寄存器中。Linux PowerPC 在进行进程的上下文切换时,将保存在进程描述符 mm→pgd 参数的 PGD 表基地址赋值于当前进程描述符的 thread.pgdir 参数。

```
/* Data TLB Error Interrupt 源代码片段 3 */
4:
  FIND_PTE
```

这段程序根据 PGD 表的基地址,调用 FIND _ PTE 函数,寻找引发 Data TLB Error Interrupt 异常的虚拟地址所对应的 PTE 表,并在这个 PTE 表中,找到与此虚拟地址对应的 Entry。FIND _ PTE 函数主要用来完成图 7-2 所示的算法。

FIND _ PTE 函数共有两个输入参数,寄存器 r10 和 r11。寄存器 r10 存放引发 Data TLB Error Interrupt 异常的虚拟地址,r11 寄存器存放当前进程 PGD 表的基地址。在该函数返回时,将 PTE 表的相应 Entry 的数值存放到 r11 寄存器中。FIND _ PTE 函数的源代码如下:

```
#define FIND_PTE \
   rlwimi r11, r10, 12, 20, 29;        /* Create L1 (pgdir/pmd) address */    \
   lwz r11, 0(r11);                    /* Get L1 entry */    \
   rlwinm. r12, r11, 0, 0, 19;         /* Extract L2 (pte) base address */    \
   beq 2f;                             /* Bail if no table */    \
   rlwimi r12, r10, 22, 20, 29;        /* Compute PTE address */    \
   lwz r11, 0(r12);                    /* Get Linux PTE */
```

这段程序的执行流程如下:

- 首先通过计算,从 PGD 表中获得与虚拟地址对应的 Entry,并将此 Entry 中的数据保存到 r11 寄存器中。
- 从 r11 寄存器中提取相应 PTE 表的基地址,并放入 r12 寄存器中。
- 如果 PTE 表的基地址为空,则表示当前 PTE 表没有被创建,此时 TLB Error Interrupt 异常处理程序将无法完成对虚拟地址的映射。此时程序将跳转到 2f,即 data _ access 函数中,建立 PTE 表,然后对虚拟地址进行映射。
- 通过计算在 PTE 表中,获得与虚拟地址对应的 Entry,并将指向该 Entry 的指针赋予 r12 寄存器。
- 将保存在 r12 寄存器中数据存放到 r11 寄存器中。从而得到 PTE 表的相应 Entry。

DataTLBError 函数通过 FIND _ PTE 函数,获得 PTE 表的相应 Entry 后,将继续执行以下程序:

```
/* Data TLB Error Interrupt 源代码片段 3 */
   andi. r13, r11, _PAGE_PRESENT /* Is the page present? */
   beq 2f    /* Bail if not present */
```

这段程序首先判断 r11 寄存器的_ PAGE _ PRESENT 位是否有效。如果无效,则当前虚

拟地址对应的物理地址页面不在物理内存，而是在对换区中，此时程序需要跳转到 2f 中执行 data _ aceess 函数。data _ access 函数将在对换区中的数据，搬移到物理内存中。

```
/* Data TLB Error Interrupt 源代码片段 4 */
    ori  r11, r11, _PAGE_ACCESSED
    stw  r11, PTE_FLAGS_OFFSET(r12)
    /* Jump to common tlb load */
    b  finish_tlb_load
```

这段程序首先标记当前物理页面正在被使用，然后跳转到 finish _ tlb _ load 函数，finish _ tlb _ load 函数的主要作用是更新 TLB0 中的相应 Entry，完成 TLB Error Interrupt 异常处理。

finish _ tlb _ load 函数有以下几个输入参数：

- 寄存器 r10，该寄存器存放引发 Data TLB Error Interrupt 异常的虚拟地址。
- 寄存器 r11，该寄存器存放与虚拟地址对应的物理地址页面和相关的属性。
- CR 寄存器的 CR5 字段，该字段的 LT 位用来表示引发 Data TLB Error Interrupt 异常的虚拟地址是属于 Linux 核心空间还是用户空间。当 LT 位为 1 时表示该虚拟地址属于用户空间，否则该虚拟地址属于 Linux 核心空间。
- 寄存器 MAS0、MAS1、MAS2 和 MAS3，这几个寄存器在进入 Data TLB Error Interrupt 异常时被 E500 内核自动保存。

该函数的详解如下：

```
finish_tlb_load:
    /*
     * We set execute, because we don't have the granularity to
     * properly set this at the page level (Linux problem).
     * Many of these bits are software only. Bits we don't set
     * here we (properly should) assume have the appropriate value.
     */
    mfspr  r12, SPRN_MAS2 rlwimi r12, r11, 26, 27, 31   /* extract WIMGE from pte */
    mtspr  SPRN_MAS2, r12
    bge  5, 1f

    /* is user addr */
    andi. r12, r11, (_PAGE_USER | _PAGE_HWWRITE | _PAGE_HWEXEC)
    andi. r10, r11, _PAGE_USER   /* Test for _PAGE_USER */
    srwi r10, r12, 1
    or r12, r12, r10   /* Copy user perms into supervisor */
    iseleq  r12, 0, r12
    b  2f
    /* is kernel addr */
1: rlwinm r12, r11, 31, 29, 29   /* Extract _PAGE_HWWRITE into SW */
    ori  r12, r12, (MAS3_SX | MAS3_SR)
```

这段程序首先使用 r11 寄存器中获得 WIMGE 字段替换 MAS2 寄存器中的 WIMGE 字

段,然后将结果暂时存放到寄存器 r12 中。

当 LT 位为 1 时 bge 指令将调转到 1f 处开始执行。这段程序的主要作用是根据虚拟地址是属于 Linux 核心空间还是用户空间,将 r11 寄存器中有关物理页面的属性赋值到 r12 寄存器中。

```
2: rlwimi r11, r12, 0, 20, 31 /* Extract RPN from PTE and merge with perms */
  mtspr  SPRN_MAS3, r11
```

将 r11 寄存器中的 RPN 和 r12 寄存器中的物理页面的属性进行拼接,然后放入寄存器 MAS3 中。

```
tlbwe

/* Done...restore registers and get out of here. */
mfspr r11, SPRN_SPRG7R
mtcr r11
mfspr r13, SPRN_SPRG5R
mfspr r12, SPRN_SPRG4R
mfspr r11, SPRN_SPRG1
mfspr r10, SPRN_SPRG0
rfi          /* Force context change */
```

这段程序使用 tlbwe 指令,更新 TLB0 的相应 Entry,之后恢复 Data TLB Error Interrupt 异常中使用的 spr 寄存器,最后使用 rfi 指令将异常处理函数返回。

7.2 Linux PowerPC 的核心空间

PowerPC 处理器系统的物理地址空间主要由主存储器空间和外部设备空间组成。一般来说,基于 E500 内核的处理器使用 DDR-SDRAM 做为主存储器的物理介质。外部设备使用的空间包括常用的一些外部设备,如 PCI 总线设备、Local Bus 设备、CPM 以及芯片内部采用存储器映射的寄存器等。

在 Linux PowerPC 中,所有这些物理地址空间最终被映射到 Linux 系统的核心空间中。对于 32 位 PowerPC 处理器,这段空间为 KERNELBASE 以上的 1 GB 大小的空间,即 0xC000-000~0xFFFF-FFFF 这段数据区域。本节所讲述的 Linux 的核心空间,就是指这段数据区域。

在 Linux PowerPC 核心空间的管理中,有些概念涉及多内核处理器及巨型机。这些概念对于绝大多数并没有巨型机经验的读者来说,不易理解。但是这些概念不得不在此处提起,因为这部分内容的存在使得 Linux PowerPC 有别于其他系统的 Linux,如 Linux ARM 和 Linux Pentium。

在学习 Linux PowerPC 核心空间的管理时,我们首先遇到的概念是非均衡式存储器访问(Non Uniform Memory Access, NUMA)和均衡式存储器访问(Uniform Memory Access, UMA)。

NUMA 结构的处理器系统由多个 SMP 结构的处理器系统组成,其中每个 SMP 结构的处理器系统具有自己的物理存储器。NUMA 结构的典型应用是超大规模并行处理系统(Mas-

sive Parallel Processor,MPP)。对于 NUMA 结构而言,由于其存储系统分布在多个处理器系统中,因此对于在其上运行的进程来说,对物理存储器的访问是不均衡的,这种存储器访问方式被称为 NUMA。

UMA 结构的处理器系统一般由多个处理器组成,这些处理器对物理存储器的访问是均衡的,即这些处理器访问物理存储器的延时相同。SMP 结构处理器系统就是典型的 UMA 结构的处理器。

尽管 Linux PowerPC 支持 NUMA 结构,但是 Linux PowerPC 的其他模块如进程管理,中断系统等绝大多数模块并没有对 MPP 结构进行支持。如果 Linux 系统需要支持 MPP 结构的处理器系统,那么程序员需要对现有的 Linux PowerPC 做许多深层次改动。事实上对于 MPP 结构处理器系统,现有的 Linux 系统并不完善,本书对此将不做详细介绍。

Linux PowerPC 设置了 pg_data_t 结构描述 NUMA 结构的每个 SMP 结构处理器的物理存储系统,并使用 node_data 数组存放所有 pg_data_t 数据结构,node_data 数组中的每一个 Entry 用来描述一个存储器节点。

SMP 结构的处理器系统或者单处理器采用 UMA 方式进行存储器访问,这种结构的处理器系统只有一个物理存储器。但是与 NUMA 结构相同,Linux 系统也使用 pg_data_t 数据结构描述整个存储器空间。但是在 SMP 结构或者单处理器结构中,仅设置了一个 pg_data_t 类型的全局变量 contig_page_data,而不是一个数组。

该数据的详细描述见./include/linux/mmzone.h 文件。Linux 系统使用宏 NODE_DATA 访问此数据。

Linux PowerPC 中,全局变量 contig_page_data 由多个数据区域组成,如下所示:

(1) ZONE_DMA。一些只有低 24 位地址总线的外部设备进行 DMA 操作时,只能访问处理器最低的 16 MB 的地址空间。ZONE_DMA 就是用来支持这类外部设备的。Pentium 处理器使用这段数据区域。

(2) ZONE_DMA32。这段数据区域由 Andi Kleen 加入,支持访问 32 位地址总线的外部设备。Pentuim 处理器使用这段区域作为 32 位的 DMA 数据区域,用来区分 ZONE_DMA 数据区域。

(3) ZONE_NORMAL。这段数据区域是最常用的,处理器系统中绝大多数物理内存都被映射到这段空间。

(4) ZONE_HIMEM。这段数据区域管理不能被处理器直接访问的高端存储器空间。当一个处理器系统的物理存储器过大时,将有一部分存储器映射到 ZONE_HIMEM 数据区域里。

但并不是所有体系结构的处理器都需要支持这四类存储器区域。如 64 位处理器系统,由于其核心空间非常大,因此不需要 ZONE_HIMEM 这段区域;对于有些体系结构的处理器,不存在 ZONE_DMA 和 ZONE_DMA32。

基于 E500 内核的 Linux PowerPC 使用了 ZONE_DMA 和 ZONE_HIMEM 两段数据区域。这两段数据区域由许多物理页面组成。在 Linux 系统中,使用 free_area 参数保存所有这些物理页面。在 Linux PowerPC 中,pg_data_t,zone,free_area 和 page 的关系如图 7-3 所示。其中 MAX_ORDER 的默认值为 11。

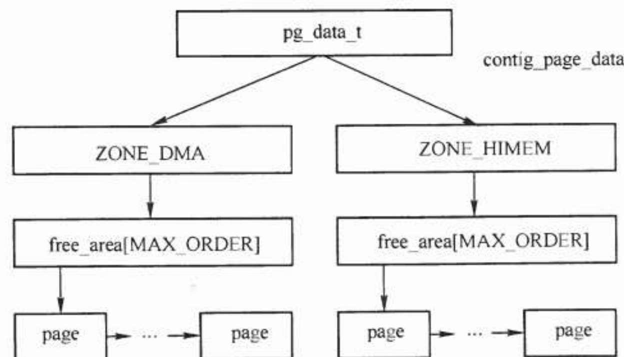


图 7-3　Linux PowerPC 中的存储节点

在图 7-3 中，每一个存储器节点由多个数据区域组成。每一个数据区域包含一个 free _ area 数组。在 free _ area 数组中，包含了一系列 page 结构，Linux 系统使用这个数组实现 Buddy System。对于 32 位处理器，Buddy System 的每一个 page 结构描述一个 4 KB 大小的物理地址空间。

7.2.1　Linux PowerPC 存储节点的数据结构

如图 7-3 所示，与 Linux PowerPC 存储节点相关的结构有 pg _ data _ t、zone、free _ area 和 page 结构。下文将对 pg _ data _ t、zone、free _ area 和 page 结构进行详细说明，而 free _ area 结构将在 7.2.3 节中介绍。

1. pg _ data _ t 结构

pg _ data _ t 结构对于单处理器系统或者 SMP 结构处理器系统的意义并不大，但是对于 MPP 结构的处理器系统十分有帮助。该结构由 IBM 结构引入，主要针对其 iSeries 和 pSeries 服务器，以及一些没有完全公开其详细设计的 MPP 结构处理器。

pg _ data _ t 结构对全机物理地址的统一编址有较大帮助。在处理器系统的实现中，将一个、四个或者是几十个处理器使用的物理地址进行统一编址也许难度并不是非常大，但是将几千个处理器进行统一编址并不是一件容易的事情。

Linux PowerPC 使用 pg _ data _ t 数据结构描述在同一个处理器系统(SMP 结构处理器或者单处理器)的所有物理内存。Linux 系统称这些 pg _ data _ t 数据结构为存储节点(Node)，该结构的详细定义在 ./include/linux/mmzone.h 文件中。

```
typedef struct pglist_data {
    struct zone node_zones[MAX_NR_ZONES];
    struct zonelist node_zonelists[GFP_ZONETYPES];
    int nr_zones;
    struct page *node_mem_map;

    struct bootmem_data *bdata;

    unsigned long node_start_pfn;
    unsigned long node_present_pages; /* total number of physical pages */
```

```
    unsigned long node_spanned_pages; /* total size of physical page range, including holes */

    int node_id;
    wait_queue_head_t kswapd_wait;
    struct task_struct * kswapd;
    int kswapd_max_order;
} pg_data_t;
```

pg_data_t结构的参数如下：

(1) node_zones参数描述当前存储器节点(Node)的数据区域类型。这些数据区域包括ZONE_HIGHMEM、ZONE_NORMAL、ZONE_DMA和ZONE_DMA32。

(2) node_zonelists参数存放Linux系统分配物理内存时的优先顺序。如果在某个数据区域内存分配失败时，将根据node_zonelists参数存放的顺序继续在其他数据区域中进行内存分配。如Linux系统，在ZONE_HIGHMEM数据区域，分配存储空间失败后，可以根据node_zonelists中规定的顺序，在ZONE_NORMAL或者ZONE_DMA数据区域中，进行内存分配。

(3) nr_zones参数保存在当前Linux系统中数据区域的个数，其最大值为4，在Linux系统中最大数据区域数为4，最小值为1。

(4) node_mem_map参数存放当前存储器节点(Node)的物理页表描述符的基地址。页表描述符是指page结构，该结构与PTE表不同。下文将详细介绍page结构。

(5) bdata参数管理Boot Memory空间。该参数和Boot Memory将在下文详细介绍。

(6) node_start_pfn参数存放当前处理器系统中，第一个物理页表的页表号PFN(Page Frame Number)。

(7) node_present_pages参数存放在当前处理器系统中使用的物理页面个数。

(8) node_spanned_pages参数存放在当前处理器系统中的全部物理页面个数。在一个处理器系统中，物理内存包含一些空洞。node_spanned_pages参数描述的物理页面个数，包括这些空洞使用的物理地址空间，而node_present_pages参数只保存实际的物理内存页面数。

举例说明，在地址范围为0x0000-0000～0x1FFF-FFFF的数据区域中，如果其0x1000-0000～0x100F-FFFF之间的空间无效(为空洞)，此时该系统的node_spanned_pages参数为131072，表示这段数据区域的总大小为512 MB，而node_present_pages参数为130816，表示这段数据区域可使用的空间为511 MB。

(9) node_id参数存放当前存储器节点(Node)的ID号。

2. zone结构

在Linux系统中，使用结构struct zone描述ZONE_NORMAL和ZONE_HIMEM、ZONE_DMA、ZONE_DMA32。该结构在./include/linux/mmzone.h文件中，其主要数据成员如下所示：

```
struct zone {
    /* Fields commonly accessed by the page allocator */
    unsigned long        free_pages;
```

```
    unsigned long      pages_min, pages_low, pages_high;
    unsigned long      lowmem_reserve[MAX_NR_ZONES];

    struct per_cpu_pageset   pageset[NR_CPUS];
    spinlock_t   lock;

    struct free_area   free_area[MAX_ORDER];

    spinlock_t   lru_lock;
    struct list_head   active_list;
    struct list_head   inactive_list;
    unsigned long      nr_scan_active;
    unsigned long      nr_scan_inactive;
    unsigned long      nr_active;
    unsigned long      nr_inactive;
    unsigned long      pages_scanned;   /* since last reclaim */
    int      all_unreclaimable; /* All pages pinned */

    struct pglist_data      *zone_pgdat;
    unsigned long      zone_start_pfn;

    char      *name;
} ____cacheline_internodealigned_in_smp;
```

(1) free_pages 参数记录在当前数据区域 Zone 内未使用页面的个数。

(2) pages_min,pages_low 和 pages_high 参数记录 Zone 内的数据阈值。当 Zone 中的可用内存较少时,kswapd 进程将被唤醒,进行页面交换。pages_min,pages_low 和 pages_high 参数用来协调 kswapd 进程进行物理内存的切换工作。

当 free_pages 小于或者等于 pages_min 时,kswapd 必须进行存储器页面交互,将一些暂时不使用的存储器空间移到 Swap 区中,此时申请物理内存页面的进程必须休眠,直到 free_pages 大于 pages_min;当 free_pages 小于等于 pages_low 时,kswapd 进程被唤醒,释放 Buddy System 中的空闲页面,以增加 free_pages 的值;pages_low 的默认值是 pages_min 的两倍;当 free_pages 大于等于 pages_high 时,kswapd 不会被唤醒,此时 Linux 系统认为当前 zone 中的内存足够大。

(3) pageset 参数存放在当前 Node 中的所有内存页面。在 Linux 中,物理页面被分为两类,hot 和 cold 类。该参数可以加快单个物理页面的申请与释放速度。

(4) lock 和 lru_lock 参数保护对 Zone 内临界数据的访问。

(5) free_area 参数是 zone 中的重要参数。这个参数对当前 Zone 的页面空间进行管理。Linux 系统使用 Buddy System 管理 Zone 内的页面空间。

(6) active_list 参数和 inactive_list 参数。Linux 系统为每一个数据区域 zone 维护 active_list 和 inactive_list 两个双向链表,这两个双向链表实现物理内存的回收。

Linux 内存管理系统将刚分配的 Cache 表项加入到 inactive_list 头部,并将其状态设置为 active。当物理内存空间有限时,需要回收 Cache 表项。此时系统首先从尾部开始反向扫描

active _ list 链表，并将状态不为 referenced 的表项加入到 inactive _ list 的头部。之后 Linux 系统继续反向扫描 inactive _ list 链表，如果所扫描的表项的处于合适的状态就回收，直到回收了足够数目的 Cache 表项。

(7) nr _ scan _ active，nr _ scan _ inactive，nr _ active 和 nr _ inactive 参数维护 inactive _ list 和 active _ list 两个双向链表的数据个数。此组参数、pages _ scanned、all _ unreclaimable 参数、active _ list 参数和 inactive _ list 参数与 Linux 内存管理的文件 Cache 有关，这部分的内容较为复杂，对此有兴趣的读者可以阅读 ./mm/vmscan.c 文件中的 shrink _ xx 函数。在一个没有 swap 区的系统中，这些参数没有太大的意义。

(8) zone _ pgdat 参数指向 pg _ data _ t 类型的指针。在 UMA 系统中，只有一个存储器节点，此时 zone _ pgdat 的值为 contig _ page _ data。

(9) zone _ start _ pfn 参数存放 Zone 中的第一个物理页表的页表号 PFN。

(10) name 参数存放 zone 的名字，如“DMA”，“Normal”或“HighMem”。

3. page 结构

在进行物理内存管理时，Linux 系统必须记录每一个物理页面的使用情况。对于 32 位处理器，Linux 系统将物理内存分为一个个独立的 4KB 页面进行管理，并设置页表描述符 page 结构，用来监控每一个物理页面的使用。在 Linux 系统中，所有页表描述符存放在全局变量 contig _ page _ data→node _ mem _ map 指针中，这个指针也被称为存储器节点的全局页表描述符指针。只有一个存储器节点的处理器系统使用 mem _ map 数组，存放这个全局页表描述符指针。

Linux PowerPC 使用 alloc _ node _ mem _ map 函数为 mem _ map 数组分配空间，使用 Boot Memory 保存 mem _ map 数组，该数组一旦被创建，在 Linux 系统的运行过程中，不会被释放。数组 mem _ map 的大小约占整个物理内存大小的 1%。在 mem _ map 中，存放着系统中所有物理页面的描述符 page。数据结构 page 的定义在 ./include/linux/mm _ types.h 文件中，其主要参数如下：

```
struct page {
    unsigned long flags;

    atomic_t _count;
    atomic_t _mapcount;

    union {
      struct {
      unsigned long private;
      struct address_space *mapping;
      };
    };
    pgoff_t index;
    struct list_head lru;
};
```

(1) flags 参数用来表示当前页面的状态，其定义在 ./include/linux/page _ flags.h 文件中。

主要状态值如表 7-3 所示。

表 7-3 page 结构 flags 参数的含义

PG _ locked	表示当前页面被锁定,不能被换出
PG _ error	表示在向此页面进行数据传递时发生错误
PG _ referenced	表示页面最近被访问过
PG _ uptodate	表示页面刚刚更新过
PG _ dirty	表示页面被改写
PG _ lru	表示页面在 active page 链表或者 inactive page 链表中
PG _ active	表示页面在 active page 链表中
PG _ slab	表示该页在 Slab 分配器中,有关 Slab 分配器的概念将会在下文中解释
PG _ checked	表示页面被一些文件系统使用
PG _ arch _ 1	表示页面不是 x86 结构下的页面
PG _ reserved	表示页面为内核代码预留或者没有使用
PG _ private	存放页面的私有数据
PG _ writeback	表示页面将被回写到外部存储器中,如硬盘
PG _ nosave	在系统挂起和恢复时使用
PG _ compound	当前页面为 compound 页
PG _ swapcache	用于 swap 区对换
PG _ mappedtodisk	表示此页面对应的数据在外部存储器中
PG _ reclaim	表示页面将被写入外部存储器中
PG _ nosave _ free	在系统挂起和恢复时使用
PG _ buddy	表示此页面在 Buddy System 中。有关 Buddy System 的概念将会在下文中解释

(2) _ count 参数存放当前页面的引用计数。该参数表示当前物理页面被多少个进程引用,即当前有多少个进程共享此物理页面。

(3) _ mapcount 的值表示有几个 PTE 表与此 page 结构关联。在 Linux PowerPC 中,不同的虚拟地址可以映射到同一个物理地址上,以共享物理内存。

(4) private 参数存放一些私有数据。

(5) mapping 参数支持 page cache 系统。

(6) lru 参数用来保存最近被使用的页表。

Linux 系统使用 mem _ init 函数,初始化当前处理器系统的所有页表描述符,本书将在下文详细介绍该函数。

7.2.2 Linux PowerPC 核心空间的初始化

Linux PowerPC 核心空间的初始化包括对全局变量 contig _ page _ data,Zone 和页表描述符的初始化。在 Linux PowerPC 中,包含两类 Zone:ZONE _ DMA 和 ZONE _ HIMEM。Linux PowerPC 使用 paging _ init 函数初始化所 Linux PowerPC 内核使用的有效地址空间。

paging _ init 函数在 Linux 系统进行初始化时,由 setup _ arch 函数调用,该函数的源代码

在 ./arch/powerpc/mm/mem.c 文件中。paging _ init 函数首先对 Zone 空间进行初始化，之后调用 free _ area _ init _ node 函数，其调用关系如图 7-4 所示。

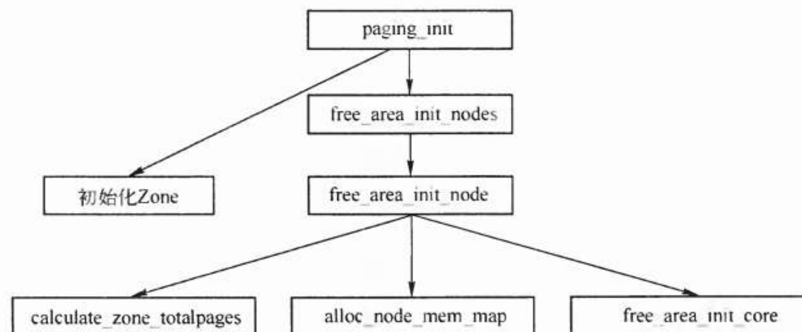


图 7-4　paging _ init 函数

paging _ init 函数主要由 free _ area _ init _ nodes->free _ area _ init _ node 函数完成，该函数首先对 Zone 空间进行初始化，之后依次调用图 7-4 中相应的函数。

1. Zone 的初始化

在 Linux PowerPC 中，Zone 空间初始化的源代码如下所示：

```
void __init paging_init(void)
{
  unsigned long start_pfn, end_pfn;
  unsigned long max_zone_pfns[MAX_NR_ZONES];
#ifdef CONFIG_HIGHMEM
   map_page(PKMAP_BASE, 0, 0); /* XXX gross */
   pkmap_page_table = pte_offset_kernel(pmd_offset(pgd_offset_k
         (PKMAP_BASE), PKMAP_BASE), PKMAP_BASE);
   map_page(KMAP_FIX_BEGIN, 0, 0); /* XXX gross */
   kmap_pte = pte_offset_kernel(pmd_offset(pgd_offset_k
         (KMAP_FIX_BEGIN), KMAP_FIX_BEGIN), KMAP_FIX_BEGIN);
   kmap_prot = PAGE_KERNEL;
#endif /* CONFIG_HIGHMEM */
```

如果处理器系统支持 HIMEM，paging _ init 函数使用 map _ page 函数为 PKMAP _ BASE 之后的虚拟地址建立 pte 表和 kmap 需要使用的 pte 表。在这些表项建立之后，处理器系统才可以访问 HIMEM 管理的物理内存。

```
/* All pages are DMA-able so we put them all in the DMA zone. */
start_pfn = __pa(PAGE_OFFSET) >> PAGE_SHIFT;
end_pfn = start_pfn + (total_memory >> PAGE_SHIFT);
add_active_range(0, start_pfn, end_pfn);

memset(max_zone_pfns, 0, sizeof(max_zone_pfns));
```

这段程序确定 Linux 系统使用的 start _ pfn(Linux 内核使用的开始页表号)和 end _ pfn

(Linux 内核使用的最后一个页表号)，然后调用 add _ active _ range 函数将这些信息保存在全局数组 early _ node _ map 中。

early _ node _ map 数组将在 free _ area _ init _ nodes 函数中使用，该数组存放 Linux 系统最初的主存储器映像。

```
#ifdef CONFIG_HIGHMEM
    max_zone_pfns[ZONE_DMA] = total_lowmem >> PAGE_SHIFT;
    max_zone_pfns[ZONE_HIGHMEM] = top_of_ram >> PAGE_SHIFT;
#else
    max_zone_pfns[ZONE_DMA] = top_of_ram >> PAGE_SHIFT;
#endif
    free_area_init_nodes(max_zone_pfns);
} /* End paging_init */
```

这段程序计算在 Linux PowerPC 中，ZONE _ DMA 和 ZONE _ HIMEM 的数据区域的大小。在 Linux PowerPC 中，只设置了 ZONE _ DMA 和 ZONE _ HIMEM 两种类型的空间。其中 ZONE _ DMA 区域存放低端内存，而 ZONE _ HIGHMEM 存放高端内存。

有些 32 位 PowerPC 处理器系统，由于使用的物理内存较大，必须使用 ZONE _ DMA 和 ZONE _ HIMEM 区域。在 Linux PowerPC 中，ZONE _ DMA 只能管理 0～768 MB 之间的物理内存，768 MB 之上的内存需要使用 ZONE _ HIMEM 区域进行管理。ZONE _ HIMEM 管理的内存不能被直接访问，需要进行空间映射，然后进行复制后才能使用。

Linux PowerPC 没有使用 ZONE _ NORMAL 这段数据区域。从数据结构的角度上看，ZONE _ DMA 和 ZONE _ NORMAL 数据区域没有什么本质的不同。在 Linux PowerPC 中，没有将 ZONE _ DMA 和 ZONE _ NORMAL 分离的必要，这是由 PowerPC 处理器的体系结构决定的。Linux PowerPC 选择使用 ZONE _ DMA 数据区域管理整个物理内存，实际上选择 ZONE _ NORMAL 数据区域管理整个物理内存也是可以的。

top _ of _ ram 参数记录内存的最大地址；total _ ram 参数记录内存的总数；total _ lowmem 参数记录低端内存的最大值，其值由 .config 文件的 CONFIG _ LOWMEM _ SIZE 参数决定。对于 Linux PowerPC，CONFIG _ LOWMEM _ SIZE 的默认值为 0x3000-0000，即 768 MB。当 Linux PowerPC 不使用 HIMEM 内存时，整个处理器系统最多支持 768 MB 物理内存。此时 ZONE _ DMA 区域的大小为 top _ of _ ram 参数中的页面个数。

Linux PowerPC 在 Zone 初始化完毕后，调用 free _ area _ init _ nodes→free _ area _ init _ node 函数，初始化当前存储器节点的相关数据结构。

```
void __meminit free_area_init_node(int nid, struct pglist_data *pgdat,
        unsigned long *zones_size, unsigned long node_start_pfn,
        unsigned long *zholes_size)
{
    pgdat->node_id = nid;
    pgdat->node_start_pfn = node_start_pfn;
    calculate_node_totalpages(pgdat, zones_size, zholes_size);
```

```
    alloc_node_mem_map(pgdat);

    free_area_init_core(pgdat, zones_size, zholes_size);
}
```

free_area_init_node 函数在 ./mm/page_alloc.c 文件中，该函数 pgdat 参数进行一些必要的初始化后，依次调用 calculate_node_totalpages、alloc_node_mem_map 和 free_area_init_core 函数。该函数有 5 个参数，分别为 nid、pgdat、zones_size、node_start_pfn 和 zholes_size。

(1) nid 参数记录当前存储器节点使用的 ID 号。对于只有一个存储器节点的处理器系统，该参数为 0；具有多个存储器节点的处理器系统，可以使用此参数实现全机物理地址的统一编址。

(2) pgdat 参数存放当前存储器节点(Node)的描述符 pg_data_t。在只有一个存储器节点的处理器系统中，定义了一个存储器节点(Node)描述符 contig_page_data。在 Linux PowerPC 中使用宏定义 NODE_DATA 访问 contig_page_data。

(3) zones_size 指针存放各个 Zone 的大小。

(4) zholes_size 指针存放各个 Zone 中空洞的大小。在一个处理器系统中，物理地址不一定连续，zones_size 参数记录存储器节点中的空洞。

(5) node_start_pfn 参数存放用来存放当前存储器节点的第一个物理页表的页表号 PFN。

2. calculate_node_totalpages 函数

calculate_zone_totalpages 函数有 3 个参数，其意义与 free_area_init_node 函数的参数意义相同，该函数源代码如下所示：

```
static void __init calculate_node_totalpages(struct pglist_data *pgdat,
        unsigned long *zones_size, unsigned long *zholes_size)
{
    unsigned long realtotalpages, totalpages = 0;
    enum zone_type i;
```

calculate_zone_totalpages 函数计算在一个存储器节点中的总页面数 totalpages 与有效页面数 realtotalpages。由于空洞的存在，在一个存储器系统中，总页面数 totalpages 与有效页面数 realtotalpages 可能不相同。该函数将存储器系统中，总页面数 totalpages 与有效页面数 realtotalpages，分别保存在 pgdat 的 node_spanned_pages 和 node_present_pages 参数中。

```
    for (i = 0; i < MAX_NR_ZONES; i++)
        totalpages += zone_spanned_pages_in_node(pgdat->node_id, i,
                        zones_size);
    pgdat->node_spanned_pages = totalpages;

    realtotalpages = totalpages;
    for (i = 0; i < MAX_NR_ZONES; i++)
```

```
        realtotalpages -=
            zone_absent_pages_in_node(pgdat->node_id, i,
                        zholes_size);
    pgdat->node_present_pages = realtotalpages;
    printk(KERN_DEBUG "On node %d totalpages: %lu\n", pgdat->node_id,
                realtotalpages);
}
```

3. alloc_node_mem_map 函数

alloc_node_mem_map 函数只有一个参数，pgdat。该函数为所有页表描述符分配空间，并将全局页表描述符指针保存在 pdgat->node_mem_map 参数中，最后将 node_mem_map 赋值到全局变量 mem_map 中，该函数的源代码如下：

```
static void __init alloc_node_mem_map(struct pglist_data *pgdat)
{
    /* Skip empty nodes */
    if (!pgdat->node_spanned_pages)
        return;
#ifdef CONFIG_FLAT_NODE_MEM_MAP
    /* ia64 gets its own node_mem_map, before this, without bootmem */
    if (!pgdat->node_mem_map) {
        unsigned long size, start, end;
        struct page *map;

        start = pgdat->node_start_pfn & ~(MAX_ORDER_NR_PAGES - 1);
        end = pgdat->node_start_pfn + pgdat->node_spanned_pages;
        end = ALIGN(end, MAX_ORDER_NR_PAGES);
        size = (end - start) * sizeof(struct page);
        map = alloc_remap(pgdat->node_id, size);
        if (!map)
            map = alloc_bootmem_node(pgdat, size);
            pgdat->node_mem_map = map + (pgdat->node_start_pfn - start);
    }
#ifdef CONFIG_FLATMEM
    if (pgdat == NODE_DATA(0)) {
        mem_map = NODE_DATA(0)->node_mem_map;
#ifdef CONFIG_ARCH_POPULATES_NODE_MAP
        if (page_to_pfn(mem_map) != pgdat->node_start_pfn)
            mem_map -= pgdat->node_start_pfn;
#endif /* CONFIG_ARCH_POPULATES_NODE_MAP */
    }
#endif
#endif /* CONFIG_FLAT_NODE_MEM_MAP */
}
```

该函数首先进行参数检查,判断当前存储器节点(Node)是否为空,如果为空则直接返回。之后这段程序通过 pgdat 中的 node_start_pfn 和 node_spanned_pages 参数,计算当前在存储器节点中,一共有多少个页面。最后使用 alloc_bootmem_node 函数分配页面描述符使用的空间,分配的空间大小为当前内存节点中的总页面数乘以 sizeof(struct page)。

当 paging_init 函数执行时,只有 Boot Memory 空间被初始化完毕,因此页表描述符只能使用 Boot Memory 中的空间。具有多个存储器节点的处理器系统,有多个页表描述符,Linux PowerPC 使用 node_mem_map 数组,保存当前处理器系统的所有页表描述符的基地址;只有一个存储器节点的处理器系统使用全局变量 mem_map 保存页表描述符的基地址。

4. free_area_init_core 函数

free_area_init_core 函数初始化 Zone 的主要参数和所有页表描述符。该函数共有 3 个参数,分别为 pgdat、zones_size 和 zholes_size,其含义与 free_area_init_node 函数的参数相同。该函数的主要源代码如下:

```
static void __meminit free_area_init_core(struct pglist_data *pgdat,
        unsigned long *zones_size, unsigned long *zholes_size)
{
...
    for (j = 0; j < MAX_NR_ZONES; j++) {
        struct zone *zone = pgdat->node_zones + j;
        unsigned long size, realsize;

        realsize = size = zones_size[j];
        if (zholes_size)
            realsize -= zholes_size[j];

        if (j < ZONE_HIGHMEM)
            nr_kernel_pages += realsize;
        nr_all_pages += realsize;

        zone->spanned_pages = size;
        zone->present_pages = realsize;
...
        zonetable_add(zone, nid, j, zone_start_pfn, size);
        ret = init_currently_empty_zone(zone, zone_start_pfn, size);
        BUG_ON(ret);
        zone_start_pfn += size;
    }
...
} /* End free_area_init_core */
```

free_area_init_core 函数首先初始化当前存储器节点的所有数据区域 Zone,并将 Zone 结构的参数进行赋值,之后使用 zonetable_add 函数将这些参数存入全局变量 zone_table 中,最后调用 init_currently_empty_zone 函数,init_currently_empty_zone 函数的源代码如下:

```
__meminit int init_currently_empty_zone(struct zone *zone,
            unsigned long zone_start_pfn,
            unsigned long size)
{
    struct pglist_data *pgdat = zone->zone_pgdat;
    int ret;
    ret = zone_wait_table_init(zone, size);
    if (ret)
      return ret;
    pgdat->nr_zones = zone_idx(zone) + 1;
    zone->zone_start_pfn = zone_start_pfn;
    memmap_init(size, pgdat->node_id, zone_idx(zone), zone_start_pfn);
    zone_init_free_lists(pgdat, zone, zone->spanned_pages);

    return 0;
}
```

init_currently_empty_zone 函数的执行流程如下：

(1) 调用 zone_wait_table_init 函数，初始化当前(Zone)的 wait table。

(2) 调用 memmap_init 函数，初始化存储器节点(Node)的每一个页面描述符。

(3) 调用 zone_init_free_lists 函数，初始化存储器节点(Node)的 Buddy System。

7.2.3 Linux PowerPC 核心空间内存的分配与释放

Linux PowerPC 使用物理页面管理其核心空间。对于 PowerPC E500 内核，Linux 系统使用的物理页面大小为 4 KB，其中每个页面都与一个 page 结构相对应。进行内存申请时，Linux 内核以一个页面为基本单位。

Linux PowerPC 的核心空间被分解为一个个的数据区域 Zone，如数据区域 ZONE_DMA 和 ZONE_HIMEM。这些数据区域由一系列物理页面组成，Linux 系统采用 Buddy System 管理在 Zone 中所有的物理内存页面。

Buddy System 属于动态存储管理技术。Buddy System 使用 Buddy 算法，实现动态存储系统管理。在一个操作系统中，除了可以使用 Buddy 算法外，还可以采用其他动态存储管理策略，如顺序搜索、分类搜索、索引搜索及位图搜索等。对于不同的应用，这些算法各有优劣。在 Linux 系统中，使用 Buddy 算法以减少整个内存区的碎片，以及整个物理内存系统的使用效率。

在 Linux 系统的运行过程中，进程会不断地申请和释放页面，因此会在一个物理地址连续的内存中形成多个空洞。有时用户需要申请一段物理连续的空间时，用户可能得不到这段内存空间，这将导致内存申请失败。而此时整个系统的可用内存数有可能大于用户所申请的内存大小，只是没有一段物理地址连续的空间大于用户所申请的空间。

在这种情况下，Linux 系统的内存管理系统需要将整个物理内存空间进行搬移，并整合以满足这段内存的申请。在一个系统中不论采用何种动态内存管理算法，这种内存的搬移和整合都无法避免，只是采用不同的算法可以决定整个内存的使用效率。

如果一个动态物理内存管理系统能够满足不同用户的各类内存申请，而不需要进行内存的整合和搬移，这种动态物理内存管理系统将是完美的系统，只是这种动态物理内存管理系统实际上并不存在。任何一个动态物理内存管理系统都存在着各种各样的约束。

Linux 系统可以支持从巨型机到手持式设备的应用，而采用的物理内存管理算法都是 Buddy 算法。虽然 Buddy 算法可以支持许多种应用，但实际上，并不能面面俱到。有能力的用户可以针对自己的应用选择不同的物理内存管理算法。

采用不同的物理内存管理算法将极大影响一个系统的实时性。一个过分要求实时性的应用往往会选择静态存储管理技术。如果一个实时操作系统使用动态存储管理技术，这种操作系统一定会对用户进行某种约束，以满足其实时性的要求。

当用户书写应用程序，驱动程序时要注意这些约束。许多系统的响应速度低，效率低下的问题往往由一些微不足道的小问题引起，只是到了系统设计的最后阶段这些小问题很难被发现。重视细节往往是一个工程师需要具备的品质，这一品质与一个工程师是否细心没有本质联系。工程师需要进行不断积累，才可能获得这一品质。

1. Linux PowerPC 空闲块的组成

Linux PowerPC 采用 Buddy 算法进行动态物理内存的管理，Buddy 算法可以操纵的最小数据单元为一个物理页面。在 Linux PowerPC 中，共使用了 ZONE _ DMA 和 ZONE _ HIMEM 两种类型的 Zone。由上文 Zone 的数据结构可知，在每个 Zone 中都有一个 free _ area 结构，这个结构记录所有在当前 Zone 中空闲物理内存页面，该结构在 Linux 系统中的定义如下所示：

```
struct free_area    free_area[MAX_ORDER]
```

其中 free _ area 参数是一个数组指针，用来保存(MAX _ ORDER + 1)个指向 free _ area 结构的指针，在 Linux PowerPC 中 MAX _ ORDER 的值为 11。在 Linux 系统中，使用这一数组指针来实现 Buddy 算法，该指针结构如图 7-5 所示。

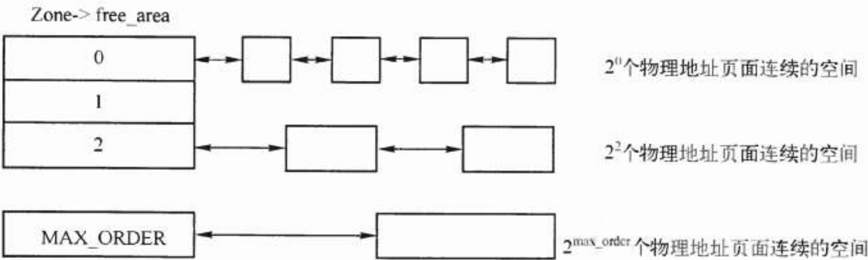


图 7-5　空闲物理内存组成结构

在 Linux 系统中，Buddy 算法将物理内存中的页面按照大小，分别进行管理。Buddy 算法将单页面(在一个页面内部，物理地址一定连续)以双向链表的方式存放在 free _ area[0]中，将两个物理地址连续的页面存放在 free _ area[1]中，并依次类推，将 2^{max_order}个物理地址连续的页面存放在 free _ area[MAX _ ORDER]中。

数据结构 free _ area 的定义在 ./include/linux/mmzone.h 文件中。

```
struct free_area {
    struct list_head free_list;
    unsigned long   nr_free;
};
```

在此结构中，free_list 参数指向物理地址连续页面所组成的双向链表，而 nr_free 参数记录这些双向链表中的成员个数。如图 7-5 所示，在 Buddy System 中，物理地址连续的最大空间为 2^{max_order} 个页面大小。在基于 32 位处理器系统的 Linux PowerPC 中，这个值为 2048 个页面，即 8 MB。

提醒读者注意，在 Linux 系统中，物理地址连续的空间资源十分有限。对于绝大多数应用，物理地址连续的 8 MB 空间已经足够。如果某些应用使用的物理地址连续的空间超过 8 MB，那么程序员需要修改 Boot Memory 的代码，将这段空间从 Boot Memory 中预留出来，使这段物理地址连续的空间不参加 Buddy System 的内存管理。使用这种方法可以申请超过 8 MB 大小的物理地址连续的空间，但是希望程序员最好使用更为巧妙的方法，避免这类物理内存的申请。

下文以一个实例说明 Buddy 算法如何对内存进行管理。当用户申请一个物理地址连续的 17 个页面空间时，由于 $2^4 < 17 < 2^5$，因此 Buddy 算法将首先检查在 free_area[5]中是否存在空闲页面，如果有则从 free_area[5]中，摘取一个空闲块分配给此用户，同时将 free_area[5]中的剩余空间 32-17 = 15 依次加入到对应的 free_area[0～4]中。

$15 = 2^0 + 2^1 + 2^2 + 2^3$，因此在 free_area[5]中分配剩余的 15 个物理页面空间，分别加入到 free_area0，free_area1，free_area2 ，free_area3，和 free_area4 中。如果当前 Linux 系统没有在 free_area[5]中发现空闲的物理页面，Buddy System 依次寻找 free_area[6～11]，直到找到具有空闲页面的 free_area 指针，将这个空闲块分配给用户，最后将剩余空间依次放到相应的 free_area 中。

如果 Buddy System 在 free_area[MAX_ORDER]中也没有找到空闲块，此时 Buddy System 必须启动 kswapd 进程进行内存整理。在这个整理过程中，将不可避免地改变 Linux 系统的 PTE 表，因此 kswapd 进程还会对 PowerPC E500 内核的 TLB0 进行刷新。更糟糕的是在一过程中，外部中断将会被禁止，从而极大地影响整个 Linux 系统的效率。幸运的是，这种情况在整个 Linux 系统中，出现的概率相当低。

物理地址连续空间的释放是申请的逆过程，当释放一个物理地址连续的 23 个页面时，Buddy System 将这段空间分别释放到 free_area[2]中。提醒读者注意，Buddy System 在物理空间释放时，会进行空闲块的合并重组，将互为伙伴关系的空闲块组成一个更大的物理地址连续的空间，放到 free_area 的相应队列中。

为了提高内存管理的效率，这种空闲块的合并重组在绝大多数情况下，并不会立即进行。我们可以猜想，如果这种空闲块的重组立即执行，即使 Linux 系统只是释放一个页面，也可能引发连锁反应。在最恶劣的情况下，整个 Buddy System 共需要进行 MAX_ORDER 次内存重组。

综上所述，使用 Buddy System 对大块物理地址连续空间的进行申请与释放时，在最恶劣的情况下，有可能极大地影响 Linux 系统的效率。因此再次提醒读者注意，在进行系统程序设

计时，需要尽量避免大块物理地址连续空间的申请与释放。

在 Linux 系统中，使用最频繁的操作是对一个物理页面的申请与释放，Linux 系统对此进行了一些特殊地处理。在 Linux 系统应用空间执行的进程，其使用的内存空间没有必要物理地址连续，使用 vmalloc 分配的内存也不需要物理地址连续。大多数核心进程、驱动程序及 Linux 内核申请的空间，也不需要多个物理地址连续的页面，真正需要多个物理页面连续的应用非常少。

2. 物理地址连续页面的申请

Linux 系统使用 alloc_page 和 alloc_pages 函数，申请物理地址连续的页面。alloc_page 函数申请一个物理页面，而 alloc_pages 函数申请多个物理地址连续的页面。在 Linux 系统中，alloc_page 函数将调用 alloc_pages 函数，因此在下文主要介绍 alloc_pages 函数。

alloc_pages 函数在 ./include/linux/gfp.h 文件中。在 alloc_pages 函数返回时，获得系统分配的物理地址连续的页面的地址。在绝大多数情况下，alloc_pages 函数不会执行失败。只有在当前系统中没有可用内存时，alloc_pages 函数才会执行失败，此时 Linux 系统将当前进程使用的栈段和整个系统的内存使用清单打印出来。

alloc_pages 函数和 alloc_page 函数的源代码如下所示。

```
#define alloc_page(gfp_mask) alloc_pages(gfp_mask, 0)
static inline struct page * alloc_pages(unsigned int gfp_mask, unsigned int order)
```

alloc_page 函数调用 alloc_pages 函数，其 order 参数为 0。alloc_pages 函数的执行流程如图 7-6 所示。

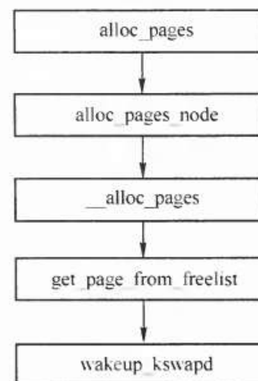


图 7-6 alloc_pages 执行流程

alloc_pages 函数有两个输入参数，分别是 gfp_mask 和 order。

(1) gfp_mask 参数。gfp_mask 参数描述所申请页面的属性。这些属性包括申请的空间属于哪个 Zone、进行空间申请时是否阻塞当前进程等一系列属性。

这些属性在 ./include/linux/gfp.h 文件中，主要属性有 GFP_NOWAIT, GFP_ATOMIC, GFP_NOIO, GFP_NOFS, GFP_KERNEL, GFP_USER, GFP_HIGHUSER, GFP_DMA, GFP_DMA32 等等，用户也可以根据实际需要对 __GFP 开头的宏进行组合获得一些特殊的 gfp_mask，不过这种机会非常少。

(2) order 参数。order 参数用来描述申请空间的大小，当 order 参数为 0 时，表示所申请的空间为一个物理页面，为 1 时表示所申请的空间为两个物理页面，并以此类推。order 参数的最大值为 MAX_ORDER。

如图 7-6 所示，alloc_pages 函数将调用一系列函数。其中 __alloc_pages 函数最为重要，该函数使用 Buddy 算法，在当前数据区域 Zone 的 free_area 数组中，找到合适的物理地址连续的空间。该函数的源代码在 ./mm/page_alloc.c 文件中，其源代码详解如下：

```
struct page * fastcall
__alloc_pages(gfp_t gfp_mask, unsigned int order, struct zonelist * zonelist)
```

```
}
  const gfp_t wait = gfp_mask & __GFP_WAIT;
  struct zone **z;
  struct page *page;
  struct reclaim_state reclaim_state;
  struct task_struct *p = current;
  int do_retry;
...
restart:
  z = zonelist->zones; /* the list of zones suitable for gfp_mask */

  if (unlikely(*z == NULL)) {
    /* Should this ever happen?? */
    return NULL;
  }

  page = get_page_from_freelist(gfp_mask|__GFP_HARDWALL, order,
          zonelist, ALLOC_WMARK_LOW|ALLOC_CPUSET);
  if (page)
    goto got_pg;
```

这段代码首先检查当前使用的数据区域 Zone。如果这段数据区域有效,则调用 get_page_from_freelist 函数,使用 Buddy 算法,在图 7-5 所示的空闲物理内存池中分配合适的空间。如果找到合适的物理页面,则跳转到 got_pg 标签处执行,否则执行以下程序:

```
...
  for (z = zonelist->zones; *z; z++)
    wakeup_kswapd(*z, order);
```

如果 get_page_from_freelist 函数没有找到合适的物理页面,则 page 参数为 NULL,表示在 free_area[MAX_ORDER]数组中没有大小合适的物理页面。此时 Linux 内核将唤醒 kswapd 监护进程,对物理内存进行碎片整理,并将 Buddy System 中互为伙伴关系的空闲物理页面组成一个更大的页面。

```
alloc_flags = ALLOC_WMARK_MIN;

if ((unlikely(rt_task(p)) && !in_interrupt()) || !wait)
  alloc_flags |= ALLOC_HARDER;

if (gfp_mask & __GFP_HIGH)
  alloc_flags |= ALLOC_HIGH;

if (wait)
  alloc_flags |= ALLOC_CPUSET;
```

```
page = get_page_from_freelist(gfp_mask, order, zonelist, alloc_flags);
if (page)
  goto got_pg;
```

使用 kswapd 监护进程完成物理内存整理后，物理内存块被重新整理。完成这个操作后，Linux 系统有可能会整合出一些较大的物理地址连续内存空间，此时该函数再次调用 get_page_from_freelist 函数，在空闲物理内存池中分配合适的空间。

```
rebalance:
...
  cond_resched();
...
  did_some_progress = try_to_free_pages(zonelist->zones, gfp_mask);
...
  if (likely(did_some_progress)) {
    page = get_page_from_freelist(gfp_mask, order, zonelist, alloc_flags);
    if (page)
      goto got_pg;
  } else if ((gfp_mask & __GFP_FS) && !(gfp_mask & __GFP_NORETRY)) {
     page = get_page_from_freelist(gfp_mask|__GFP_HARDWALL, order,
         zonelist, ALLOC_WMARK_HIGH|ALLOC_CPUSET);
  if (page)
    goto got_pg;

  out_of_memory(zonelist, gfp_mask, order);
  goto restart;
}
```

如果使用 kswapd 监护进程完成物理内存整理后，get_page_from_freelist 函数仍然不能在空闲物理内存池中找到大小合适的空间，则调用 try_to_free_pages 函数，继续释放一些物理内存，之后再次调用 get_page_from_freelist 函数，在空闲物理内存池中分配合适的空间。如果此时 get_page_from_freelist 函数返回值依然为 NULL，则调用 out_of_memory 函数，表示在当前空闲物理内存池中没有合适的物理空间。

Linux PowerPC 支持另外一组函数申请物理连续的页面空间，分别为__get_free_page 函数、__get_free_pages 函数和__get_dma_pages 函数。这组函数的入口参数与 alloc_pages 函数的入口相同。在这组函数中，将调用 alloc_pages 函数申请物理地址连续的页面空间。这组函数与 alloc_pages 函数不同之处在于函数返回值，这组函数的返回值为物理内存的虚拟地址。

3. 物理地址连续页面的释放

物理地址连续页面的释放是申请的逆过程。在 Linux PowerPC 中，使用__free_pages 函数、__free_page 函数、free_pages 函数和 free_page 函数完成这一过程。这些函数的原型定义如下所示。

```
void free_pages(unsigned long addr, unsigned int order)
#define free_page(addr) free_pages((addr),0)

void __free_pages(struct page *page, unsigned int order)
#define __free_page(page) __free_pages((page), 0)
```

free_pages 函数和 free_page 函数用于物理页面的释放，是__get_free_pages 函数和__get_free_page 函数的逆过程，其入口参数 addr 为虚拟地址。而__free_pages 函数和__free_pages 函数与 alloc_pages 函数和 alloc_page 函数相对应，其入口参数 page 为指向页表的指针。这些与释放物理页面有关的函数调用关系如图 7-7 所示。

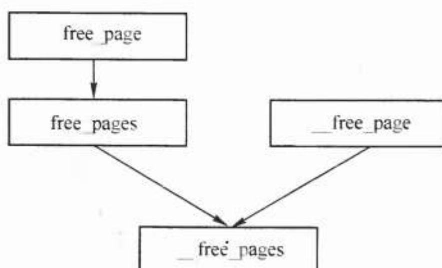


图 7-7　free_page 系列函数调用关系图

由上图所示，free_page 系列函数最终调用__free_pages 函数，完成物理地址连续的页面释放，__free_pages 函数的定义在 ./mm/page_alloc.c 文件中，该函数的源代码如下所示：

```
fastcall void __free_pages(struct page *page, unsigned int order)
{
  if (put_page_testzero(page)) {
    if (order == 0)
      free_hot_page(page);
    else
      __free_pages_ok(page, order);
  }
}
```

__free_pages 函数首先调用 put_page_testzero 函数，检查当前页表描述符 page 的引用计数，并将__count 减一。如果__count 的值不为零，表示在当前系统中，还有其他程序使用当前物理页面，此时__free_pages 函数将直接返回；否则该函数调用 free_hot_page 函数或者__free_pages_ok 函数释放物理页面。

__free_pages 函数调用 free_hot_page 函数释放一个物理页面，否则调用__free_pages_ok 函数。在 Linux 系统中，最经常使用的是对一个物理页面的申请和释放操作。为此，Linux 系统采用了一些优化措施，本书将在下文介绍这些内容。

如果__free_pages 函数释放多个页面，将调用__free_pages_ok 函数。__free_pages_ok 函数在 ./mm/page_alloc.c 文件中，该函数调用 free_one_page 函数，释放多个物理地址连续的页面。在多个物理地址连续页面的释放过程中，Buddy System 将对物理页面进行整

合，将互为 Buddy 关系的物理页面，整合为一个更大的物理页面。

Linux 系统调用 free_one_page 函数时，需要使用 local_irq_save 函数和 local_irq_restore 函数进行保护，因此在执行 free_one_page 函数将不允许外部中断。

Buddy 算法进行页面合并时，在最恶劣的情况下，耗费的时间较长，因此这种做法将在某种程度上影响系统的响应速度。__free_pages_ok 函数的源代码如下所示：

```
static void __free_pages_ok(struct page * page, unsigned int order)
{
  unsigned long flags;
  int i;
  int reserved = 0;
...
  local_irq_save(flags);
  __count_vm_events(PGFREE, 1 << order);
  free_one_page(page_zone(page), page, order);
  local_irq_restore(flags);
}
```

__free_pages_ok 函数调用 free_one_page->__free_one_page 函数，完成多个物理地址连续页面的释放，__free_one_page 函数的主要源代码如下所示：

```
static inline void __free_one_page(struct page * page,
    struct zone * zone, unsigned int order)
{
  unsigned long page_idx;
  int order_size = 1 << order;

  if (unlikely(PageCompound(page)))
    destroy_compound_page(page, order);

  page_idx = page_to_pfn(page) & ((1 << MAX_ORDER) - 1);

  VM_BUG_ON(page_idx & (order_size - 1));
  VM_BUG_ON(bad_range(zone, page));

  zone->free_pages += order_size;
  while (order < MAX_ORDER - 1) {
    unsigned long combined_idx;
    struct free_area * area;
    struct page * buddy;

    buddy = __page_find_buddy(page, page_idx, order);
    if (! page_is_buddy(page, buddy, order))
      break;      /* Move the buddy up one level. */
```

```
        list_del(&buddy->lru);
        area = zone->free_area + order;
        area->nr_free--;
        rmv_page_order(buddy);
        combined_idx = __find_combined_index(page_idx, order);
        page = page + (combined_idx - page_idx);
        page_idx = combined_idx;
        order++;
    }
    set_page_order(page, order);
    list_add(&page->lru, &zone->free_area[order].free_list);
    zone->free_area[order].nr_free++;
}
```

__free_one_page 函数是多个物理地址连续页面释放的关键。此函数将依次扫描存放在 free_area[MAX_ORDER]数组中各种物理地址连续的内存页面，然后将彼此为 Buddy 关系的物理页面重组为更大的物理地址连续页面。

在最恶劣的情况下，释放一个物理地址连续的两个页面，最多要进行 MAX_ORDER-2 次页面合并，这将耗费处理器较长的时间。因此用户在编写 Linux 设备驱动程序时，应尽量避免申请以及释放多个物理地址连续的页面。

__free_one_page 函数调用__page_find_buddy 函数，寻找当前物理页面的 Buddy 页面，并使用 page_is_buddy 函数确定当前物理页面与当前页面是否互为 Buddy 关系。在这两个函数的注释中，清楚地介绍了如何确定当前物理页面的 Buddy 页面，以及如何确定物理页面的 Buddy 关系。这两个函数的注释如下：

```
/*
 * Locate the struct page for both the matching buddy in our
 * pair (buddy1) and the combined O(n+1) page they form (page).
 *
 * 1) Any buddy B1 will have an order O twin B2 which satisfies
 * the following equation:
 * B2 = B1 ^ (1 << O)
 * For example, if the starting buddy (buddy2) is #8 its order
 * 1 buddy is #10:
 * B2 = 8 ^ (1 << 1) = 8 ^ 2 = 10
 *
 * 2) Any buddy B will have an order O+1 parent P which
 * satisfies the following equation:
 * P = B & ~(1 << O)
 *
 * Assumption: *_mem_map is contiguous at least up to MAX_ORDER
 */
```

```
static inline struct page *
__page_find_buddy(struct page *page, unsigned long page_idx, unsigned int order)
{
...
}
/*
 * This function checks whether a page is free && is the buddy
 * we can do coalesce a page and its buddy if
 * (a) the buddy is not in a hole &&
 * (b) the buddy is in the buddy system &&
 * (c) a page and its buddy have the same order &&
 * (d) a page and its buddy are in the same zone.
 *
 * For recording whether a page is in the buddy system, we use PG_buddy.
 * Setting, clearing, and testing PG_buddy is serialized by zone->lock.
 *
 * For recording page's order, we use page_private(page).
 */
static inline int page_is_buddy(struct page *page, struct page *buddy,
int order)
{
...
}
```

4. 单个物理页面的申请与释放

Linux 系统使用 get_page_from_freelist 函数申请物理页面。buffered_rmqueue 函数是 get_page_from_freelist 函数的主体。buffered_rmqueue 函数能够申请单个或者多个物理地址页面连续的空间。

buffered_rmqueue 函数对单个物理页面的申请进行了额外处理。该函数申请单个物理页面时，首先从当前数据区域 Zone 的 per_cpu_pageset 参数中，直接获得所需要的物理页面，而不需要使用 Buddy 算法搜索当前数据区域 Zone 的 free_area[MAX_ORDER]内存池。当前数据区域 Zone 的 pcp(per_cpu_pageset)参数也称作单个物理页面缓冲池。

buffered_rmqueue 函数的源代码在 ./mm/page_alloc.c 文件中，如下所示：

```
static struct page *buffered_rmqueue(struct zonelist *zonelist,
        struct zone *zone, int order, gfp_t gfp_flags)
{
  unsigned long flags;
  struct page *page;
  int cold = !!(gfp_flags & __GFP_COLD);
  int cpu;

again:
```

```
        cpu = get_cpu();
        if (likely(order == 0)) {
            struct per_cpu_pages *pcp;

            pcp = &zone_pcp(zone, cpu)->pcp[cold];
            local_irq_save(flags);
            if (!pcp->count) {
                pcp->count = rmqueue_bulk(zone, 0,
                            pcp->batch, &pcp->list);
                if (unlikely(!pcp->count))
                    goto failed;
            }
            page = list_entry(pcp->list.next, struct page, lru);
            list_del(&page->lru);
            pcp->count--;
```

通过这段代码，可以发现当order等于0，即申请单个物理页面时，buffered_rmqueue函数将在当前数据区域Zone的per_cpu_pages参数中获得所需要的空间，其执行流程如下：

(1) 首先该函数将检查在当前数据区域的zone->per_cpu_pageset->pcp中，是否有空闲的物理页面，即pcp->count参数是否为0。

(2) 如果pcp->count参数为0，表示在当前数据区域Zone的pcp参数中，没有空闲的物理页面，此时需要调用rmqueue_bulk函数，从当前数据区域Zone的free_area[MAX_ORDER]中获得多个物理页面填充到pcp中同时更新pcp->count。

(3) 从pcp队列中获得所需的单个物理页面，然后将该物理页面从pcp队列中摘除，并更新pcp->count参数。pcp队列使用lru指针，将当前pcp的所有空闲物理页面组成一个双向链表。

```
        } else {
            spin_lock_irqsave(&zone->lock, flags);
            page = __rmqueue(zone, order);
            spin_unlock(&zone->lock);
            if (!page)
                goto failed;
        }
    ...
        return page;
    failed:
        local_irq_restore(flags);
        put_cpu();
        return NULL;
    } /* End buffered_rmqueue */
```

如果order不等于0，即申请多个物理地址连续的页面时，该函数将调用__rmqueue函数，

从当前数据区域 Zone 的 free_area[MAX_ORDER]中获得多个物理页面,其源代码如下所示:

```
static struct page * __rmqueue(struct zone * zone, unsigned int order)
{
  struct free_area * area;
  unsigned int current_order;
  struct page * page;

  for (current_order = order; current_order < MAX_ORDER; ++current_order) {
    area = zone->free_area + current_order;
    if (list_empty(&area->free_list))
      continue;

    page = list_entry(area->free_list.next, struct page, lru);
    list_del(&page->lru);
    rmv_page_order(page);
    area->nr_free--;
    zone->free_pages -= 1UL << order;
    expand(zone, page, order, current_order, area);
    return page;
  }

  return NULL;
}
```

__rmqueue 函数根据 Buddy 算法,搜索 free_area[order]~free_area[MAX_ORDER]间的物理页面,直到获得合适的物理页面。

Linux 系统释放单个页面时,调用 free_hot_page->free_hot_cold_page 函数。该函数的源代码详解如下:

```
static void fastcall free_hot_cold_page(struct page * page, int cold)
{
...
  list_add(&page->lru, &pcp->list);
  pcp->count++;
  if (pcp->count >= pcp->high) {
    free_pages_bulk(zone, pcp->batch, &pcp->list, 0);
    pcp->count -= pcp->batch;
  }
...
}
```

由以上源代码可知,在大多数情况下,Linux 系统释放单个物理页面时,仅需要将这个物理页面加入 pcp 中的 lru 队列中即可。只有在 pcp 中的空闲物理页面过多,即 pcp->count >= pcp->high 时,才会调用 free_pages_bulk 函数,将此物理页面按照 buddy 算法释放到 free

_ area[MAX _ ORDER]中，此时 Linux 系统才会对物理空间进行整合。

至此，我们简单介绍了如何使用 Buddy 算法进行 Linux 的动态物理内存的管理。可以发现，在最恶劣的情况下，无论 Linux 系统分配还是释放一个物理页面，都有可能引起连锁反应，从而需要大量的处理器时间。由此可以初步判定，Buddy 算法在很大程度上并不适合强实时的应用。Linux 系统在使用 Buddy System 系统进行物理内存的整合时，将关闭外部中断。此时将影响外部中断的响应速度。为此，许多公司在实现自己的“实时 Linux 系统”时，有时需要改变物理内存管理策略。

5．Buddy System 的初始化

Linux 系统共分两个步骤完成对 Buddy System 的初始化：在 setup _ arch 函数中，调用 paging _ init 函数；调用 free _ area _ init _ core ->init _ currently _ empty _ zone ->zone _ init _ free _ lists 函数，对 Buddy System 的对应数据区域 Zone 中的 free _ area 结构完成初始化，如图 7-4 所示。zone _ init _ free _ lists 函数的源代码如下所示：

```
void zone_init_free_lists(struct pglist_data *pgdat, struct zone *zone,
        unsigned long size)
{
  int order;
  for (order = 0; order < MAX_ORDER ; order++) {
    INIT_LIST_HEAD(&zone->free_area[order].free_list);
    zone->free_area[order].nr_free = 0;
  }
}
```

从以上源代码我们可以发现，zone _ init _ free _ lists 函数执行完毕后，在数据区域 zone 的 free _ area[MAX _ ORDER]中，并没有包含任何物理页面。在 free _ area 数组中，空闲页面的个数都为 0。此时的初始化，仅是对数据区域 Zone 中的 free _ area 结构进行基本的初始化。

在 Linux 系统中，mem _ init 函数对 Buddy System 进行真正意义上的初始化。该函数调用 free _ all _ bootmem->free _ all _ bootmem _ core 函数，将数据区域 Zone 中 free _ area 参数进行赋值，并将空闲的物理页面一个个地放入对应的 free _ area 链表中，完成对 free _ area 链表的初始化。mem _ init 函数在 ./arch/powerpc/mm/mem.c 文件中，该函数在 Linux 系统初始化时由 start _ kernel 函数调用。

free _ all _ bootmem _ core 函数将对每一个物理页面进行检查，如果当前物理页面没有被使用，该函数调用__ free _ pages _ bootmem ->__ free _ page 函数，将物理页面一个个地释放到 Buddy Sytem 所管理的物理内存系统中，即加入到 free _ area 链表中。

__ free _ page 函数用于释放单个物理页面。在 Linux 系统中，Buddy System 系统没有初始化完毕之前，物理内存将由 Boot Memory 管理。此时，__ free _ page 函数将释放由 Boot Memory 管理的物理页面，然后加入到 free _ area 参数中。

free _ all _ bootmem _ core 函数的源代码在 ./mm/bootmem.c 文件中，该函数将在 7.2.4 节中详细说明。Linux 系统在初始化 free _ area[MAX _ ORDER]管理的物理内存空间时，使用__ free _ page 函数将空闲物理页面放入 pcp->list 队列中的，当此队列中的 pcp->count 参数大于 pcp->high 参数时，Linux 系统使用 free _ pages _ bulk 函数将 free _ area[MAX _ ORDER]

中的物理页面进行整合。

由于在 Linux 系统初始化时，绝大多数物理页面都是物理地址连续的，因此完全没有必要将整个空间的每一个物理页面逐个加到 Buddy System 管理的物理内存中，这种做法的效率非常低。不过似乎 Linux 的维护者并没有太多的精力优化初始化部分的代码。如果用户需要缩短 Linux 系统的启动时间，可以对这一部分代码进行适当修改，以略微提高 Linux 系统的启动速度。

在 Linux 系统中，mem _ init 函数执行完毕后，Buddy 系统被初始化完毕，此时全局变量 mem _ init _ done 将被置为 1，之后 Buddy System 将开始对 Linux 系统的物理内存进行接管。在全局变量 mem _ init _ done 将被置为 1 之前，即从 Linux 系统开始引导到 mem _ init 函数结束之前，Linux 系统不能使用本节中的函数分配物理内存，而必须使用 Boot Memory 进行物理内存的申请与释放。

7.2.4 Boot Memory 分配器

Linux 系统在启动到 mem _ init 函数执行完毕的这段时间里，Buddy System 没有初始化完毕。在此阶段，Linux 系统可以直接访问的数据空间包括 Linux 内核的 data 段、bss 段及在 sdata 段中的数据。但是在 Linux 系统进行初始化时，有时不可避免地动态申请某些内存空间，因为 Linux 系统不可能将所有数据静态地链接到 Linux 内核中。在这段时间里，Linux 系统使用 Boot Memory 分配器管理物理内存。

Boot Memory 分配器管理物理内存的算法较为简单。Boot Memory 使用 FFB(First Fit Allocator)算法管理 Boot Memory 物理内存。First Fit Allocator 算法使用位图(bitmap)描述整个物理内存空间。其中，位图中的每一位对应一个实际的物理页面。如果位图中的某一位为 1 时，表示对应的物理页面已被使用，为 0 表示对应的物理页面未被使用。

与 Buddy 算法相比，这种算法十分简单。这种算法带来的不利影响是，系统程序员使用 Boot Memory 申请内存和释放内存操作时，十分容易在物理内存中留下一个个空洞。因此，在使用 First Fit Allocator 算法进行内存分配时，必须十分小心。

提醒使用 Boot Memory 分配器的程序员注意，要小心使用 Boot Memory 空间。绝大多数在系统初始化中使用的 Boot Memory，将不会被释放。

Linux 系统使用 bootmem _ data 结构，对 Boot Memory 进行描述。在上文中，可以发现在 pg _ data _ t 结构中，包含一个 bootmem _ data 结构的参数 bdata，该参数用来保存整个存储器节点的 Boot Memory 信息。

bootmem _ data 结构的定义在 ./include/linux/bootmem.h 文件中。其主要参数如下所示：

```
typedef struct bootmem _ data {
  unsigned long node _ boot _ start;
  unsigned long node _ low _ pfn;
  void * node _ bootmem _ map;
  unsigned long last _ offset;
  unsigned long last _ pos;
```

```
    unsigned long last_success;   /* Previous allocation point. To speed
            * up searching */
    struct list_head list;
} bootmem_data_t;
```

- node_boot_start 参数存放 Boot Memory 可以管理的第一个物理内存的地址，即 Boot Memory 的起始地址。这个地址在 Linux 核心空间之后。
- node_low_pfn 参数存放 Boot Memory 所能管理的最后一个物理页面的页面号。
- node_bootmem_map 参数存放 First Fit Allocator 算法所使用的物理内存位图信息，这个物理内存信息被存放在 Boot Memory 的开始处。
- last_pos 参数存放上次分配的 Boot Memory 内存，所使用的最后一个页面的 pfn。
- last_offset 参数存放上次 Boot Memory 内存分配结束后，在 last_pos 页面中的偏移。
- last_success 参数保存上次 Boot Memory 内存释放的地址。last_pos、last_offset 和此参数用来加速 Linux 内核程序对 Boot memory 空间的申请操作。

1. Boot Memory 的初始化

Linux 系统调用 setup_arch->do_init_bootmem 函数对整个 Boot Memory 进行初始化，do_init_bootmem 函数在 ./arch/powerpc/mm/mem.c 文件中。

```
void __init do_init_bootmem(void)
{
    unsigned long i;
    unsigned long start, bootmap_pages;
    unsigned long total_pages;
    int boot_mapsize;
```

do_init_bootmem 函数没有输入参数，也没有返回值，该函数使用 Linux 系统的全局变量对 Boot Memory 进行初始化。

```
    max_pfn = total_pages = lmb_end_of_DRAM() >> PAGE_SHIFT;
#ifdef CONFIG_HIGHMEM
    total_pages = total_lowmem >> PAGE_SHIFT;
#endif
```

这段程序的执行流程如下：

- 初始化 max_pfn 参数和 total_pages 参数。其中 max_pfn 参数存放在前系统中最大的物理页面号，而 total_pages 参数存放当前系统中可用的物理页面数量。Linux 系统缺省认为物理内存从零开始编址。在多数情况下，max_pfn 参数和 total_pages 参数的值相等。
- 如果 Linux 系统使用高端内存(HIGHMEM)，do_init_bootmem 函数调整 total_pages 参数为 total_lowmem>>PAGE_SHIFT，total_lowmem 参数存放 Linux 系统低端内存的大小。Linux 系统的 Boot Memory 不能管理高端内存，因此这段程序需要调正 total_pages 参数。

```
/*
 * Find an area to use for the bootmem bitmap. Calculate the size of
 * bitmap required as (Total Memory) / PAGE_SIZE / BITS_PER_BYTE.
 * Add 1 additional page in case the address isn't page-aligned.
 */
bootmap_pages = bootmem_bootmap_pages(total_pages);

start = lmb_alloc(bootmap_pages << PAGE_SHIFT, PAGE_SIZE);

boot_mapsize = init_bootmem(start >> PAGE_SHIFT, total_pages);
```

这段程序的执行流程如下：

- 首先调用 bootmem_bootmap_pages 函数，计算在当前 Linux 系统中，一共需要多少内存空间，保存 Boot Memory 的位图信息，并将这个结果存入 bootmap_pages 参数中。bootmem_bootmap_pages 函数的定义在 ./mm/bootmem.c 文件中。
- 然后调用 lmb_alloc 函数为 Boot Memory 的位图预留空间。
- 使用 init_bootmem 函数将从 0 到 total_pages 之后的空间进行初始化，该函数首先将整个 Boot Memory 的位图都置为全 1，即所有内存都已经被使用，同时规定 Boot Memory 的可用内存从 start 变量开始。

```
/* Add active regions with valid PFNs */
for (i = 0; i < lmb.memory.cnt; i++) {
  unsigned long start_pfn, end_pfn;
  start_pfn = lmb.memory.region[i].base >> PAGE_SHIFT;
  end_pfn = start_pfn + lmb_size_pages(&lmb.memory, i);
  add_active_range(0, start_pfn, end_pfn);
}

/* Add all physical memory to the bootmem map, mark each area
 * present.
 */
#ifdef CONFIG_HIGHMEM
  free_bootmem_with_active_regions(0, total_lowmem >> PAGE_SHIFT);
#else
  free_bootmem_with_active_regions(0, max_pfn);
#endif
```

这段程序的执行流程如下：

- 使用 add_active_range 函数，将当前 Linux 系统所使用的数据区域逐个加到 early_node_map 表中。
- 调用 add_active_range 函数初始化 early_node_map 表，该表存放当前存储器的映像。free_area_init_nodes 函数使用该映像，计算当前处理器系统使用的存储器的大小和存储器之间的空洞。
- 使用 free_bootmem_with_active_regions 函数，将所有在 early_node_map 表中的物

理内存再次释放，并将所有物理内存的描述信息放入 Boot Memory 的位图中，此时该位图中的所有位被清零，表示所有 Boot Memory 内存都未使用。

```
/* reserve the sections we're already using */
for (i = 0; i < lmb.reserved.cnt; i++)
  reserve_bootmem(lmb.reserved.region[i].base,
        lmb_size_bytes(&lmb.reserved, i));
```

上述程序使用 reserve_bootmem 函数在 Boot Memory 管理的物理地址空间中预留一部分空间。

在 Linux PowerPC 中，IBM 的工程师创造了 LMB(Logic Memory Block)结构，用来管理所有物理内存空间。对 LMB 有兴趣的读者，可以阅读 ./arch/powerpc/mm/lmb.c 文件和 ./include/asm-powerpc/lmb.h 文件，本书对此不做介绍。对于单处理器系统，LMB 结构没有太大用处，它主要用来管理多处理器的内存系统。

do_init_bootmem 函数将以下几个空间预留：

- Linux 内核的 text 段，data 段和 bss 段所占用的空间。
- 0～0x4000 之间的物理地址空间。许多 PowerPC 的内核，如 603E，604E，使用这段空间作为中断向量。
- initrd 所占用的物理地址空间，这段空间实现初始 RAM Disk。

```
  /* XXX need to clip this if using highmem? */
  sparse_memory_present_with_active_regions(0);

  init_bootmem_done = 1;
}
```

在 do_init_bootmem 函数的最后，将 init_bootmem_done 参数置为 1，标志整个 Boot Memory 初始化完毕。

2. Boot Memory 的分配

Linux 系统使用宏 alloc_bootmem，申请 Boot Memory 物理地址空间，宏 alloc_bootmem 的定义在 ./include/linux/bootmem.h 文件中。该函数调用图 7-8 中的函数，实现 Boot Memory 物理地址空间的分配。

宏 alloc_bootmem 最后调用__alloc_bootmem_core 函数，在 Boot Memory 中，进行物理内存分配。__alloc_bootmem_core 函数在 ./mm/bootmem.c 文件中，该函数的详解如下：

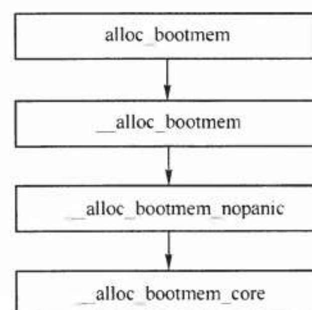


图 7-8　alloc_bootmem 的函数调用

```
void * __init
__alloc_bootmem_core(struct bootmem_data *bdata, unsigned long size,
        unsigned long align, unsigned long goal, unsigned long limit)
{
```

```
    unsigned long offset, remaining_size, areasize, preferred;
    unsigned long i, start = 0, incr, eidx, end_pfn;
    void * ret;
```

__alloc_bootmem_core 函数从 alloc_bootmem 函数中继承了一个参数 size,该参数用来描述从 Boot Memory 中申请的空间大小。使用__alloc_bootmem_core 函数申请的物理地址空间以 Cache 行对界。__alloc_bootmem_core 函数首先进行参数检查,在此过程中,将主要对 Boot Memory 的边界,如 size,align,node_boot_start 参数进行边界检查。

```
...
restart_scan:
  for (i = preferred; i < eidx; i += incr) {
    unsigned long j;
    i = find_next_zero_bit(bdata->node_bootmem_map, eidx, i);
    i = ALIGN(i, incr);
    if (i >= eidx)
      break;
    if (test_bit(i, bdata->node_bootmem_map))
      continue;
    for (j = i + 1; j < i + areasize; ++j) {
      if (j >= eidx)
        goto fail_block;
      if (test_bit(j, bdata->node_bootmem_map))
        goto fail_block;
    }
    start = i;
    goto found;
  fail_block:
    i = ALIGN(j, incr);
  }
```

这段程序使用 find_next_zero_bit 函数,对整个 Boot Memmory 的位图进行扫描,以获得合适的物理页面。find_next_zero_bit 函数使用了 PowerPC 处理器的"cntlzw"指令以加速扫描速度。"cntlzw"指令能够极大地提高位图的扫描速度。

```
...
found:
  bdata->last_success = PFN_PHYS(start);
  BUG_ON(start >= eidx);
...
  /*
   * Reserve the area now:
   */
  for (i = start; i < start + areasize; i++)
```

```
        if (unlikely(test_and_set_bit(i, bdata->node_bootmem_map)))
            BUG();
    memset(ret, 0, size);
    return ret;
} /* End __alloc_bootmem_core */
```

这段程序使用 last_pos 参数和 last_offset 参数,获得待申请空间的地址。最后更新 Boot Memory 的位图信息,将从 Boot Memory 中获得的数据空间清零。在 Buddy System 没有初始化完毕之前,Linux 系统的启动程序只能使用 Boot Memory 空间。

除此之外,用户如果需要大段物理地址连续的空间,也可以使用 alloc_bootmem 函数在 Boot Memory 中申请空间。

如果用户需要的物理地址连续的空间大于 Buddy System 所能管理的最大物理地址连续的空间时,只能在 Linux 系统启动的初期,使用 alloc_bootmem 函数申请这段物理地址连续的内存空间。在操作系统中,大段物理地址连续的空间十分珍贵,而且十分有限。

3. Boot Memory 的释放

与 Boot Memory 的申请过程相比,Boot Momory 的释放过程较为简单。Linux 系统使用 free_bootmem 函数,释放 Boot Memory 空间。在 Linux 系统中, free_bootmem 函数几乎很少被使用。一般来说,使用 alloc_bootmem 函数申请的空间,在整个 Linux 系统运行中,一直有效,系统程序员很少去释放这段空间。

free_bootmem 函数调用 free_bootmem_core 函数,完成 Boot Memory 内存空间的释放, free_bootmem_core 函数在 ./mm/bootmem.c 文件中。该函数的实现十分简单,本文将不再对这段代码进行详细介绍,绝大多数用户没有机会使用这个函数。

4. Boot Memory 的清除

Boot Memory 的清除不同于 Boot Memory 的释放。所谓 Boot Memory 的清除指将 Boot Memory 位图中没有使用的物理页面去除;而 Boot Momory 的释放指将 Boot Memory 位图中使用的物理页面去除。

Linux 系统使用 free_all_bootmem 函数,将 Boot Memory 中的物理页面清除。该函数将逐个遍历 Boot Memory 的位图信息,并分析所有物理页面。当位图中的页面没有被使用,则从 Boot Memory 中释放此物理页面,并将此页面加入到 Buddy System 管理的内存中,如果位图中的页面正在被使用,则此页面将被保留。

如果 Linux 系统在进行初始化时使用 alloc_bootmem 函数分配了一段物理内存空间,而这段空间并没有使用 free_bootmem 函数释放,那么这段物理地址空间将在 Linux 系统运行的生命周期中一直存在。free_all_bootmem 函数在 ./mm/bootmem.c 文件中。该函数没有输入参数,其返回值为从 Boot Memory 中释放的所有物理页面的总和。

free_all_bootmem 函数将调用 free_all_bootmem_core 函数。free_all_bootmem_core 函数也在 ./mm/bootmem.c 文件中,其主要的源代码如下所示:

```
static unsigned long __init free_all_bootmem_core(pg_data_t *pgdat)
{
    struct page *page;
    unsigned long pfn;
```

```
bootmem_data_t *bdata = pgdat->bdata;
unsigned long i, count, total = 0;
unsigned long idx;
unsigned long *map;
int gofast = 0;
```

free_all_bootmem_core 函数的返回值与 free_all_bootmem 函数的返回值相同，其输入参数 pgdat 为处理器系统的存储器节点指针。

```
...
    /* Check physaddr is O(LOG2(BITS_PER_LONG)) page aligned */
    if (bdata->node_boot_start == 0 ||
      ffs(bdata->node_boot_start) - PAGE_SHIFT > ffs(BITS_PER_LONG))
      gofast = 1;
    for (i = 0; i < idx; ) {
      unsigned long v = ~map[i / BITS_PER_LONG];

      if (gofast && v == ~0UL) {
        int order;

        page = pfn_to_page(pfn);
        count += BITS_PER_LONG;
        order = ffs(BITS_PER_LONG) - 1;
        __free_pages_bootmem(page, order);
        i += BITS_PER_LONG;
        page += BITS_PER_LONG;
      } else if (v) {
        unsigned long m;

        page = pfn_to_page(pfn);
        for (m = 1; m && i < idx; m<<=1, page++, i++) {
          if (v & m) {
            count++;
            __free_pages_bootmem(page, 0);
          }
        }
      } else {
        i += BITS_PER_LONG;
      }
      pfn += BITS_PER_LONG;
    }
    total += count;
```

free_all_bootmem_core 函数的主要作用是，清除在 Boot Memory 中的物理页面。在这段程序中，设计者为了提高 Boot Memory 物理页面的清除速度，使用了一个相对快速的算法。

这个算法,在某种程度上,并不完美。

- 在这段程序中,首先检查 Boot Memory 的起始地址是否为多个页面对界。对于 32 位操作系统,当起始地址为 32 个页面物理地址连续时,gofast 参数为 1。此时,该函数可以一次检查 32 位的位图信息。如果在位图中,所有 32 位都为 0,该函数将一次释放 32 个物理页面。如果在这个 32 位的位图中,有一位不为 0,则表示在这个位图描述的 32 个页面中,至少有一个页面被使用。此时,该函数将逐个检查 32 位的位图,并将未使用的物理页面释放。
- 由上文得知,__free_pages_bootmem 函数将由 Boot Memory 管理的空间加入到 Buddy System 中。

```
    /*
     * Now free the allocator bitmap itself, it's not
     * needed anymore:
     */
    page = virt_to_page(bdata->node_bootmem_map);
    count = 0;
    idx = (get_mapsize(bdata) + PAGE_SIZE - 1) >> PAGE_SHIFT;
    for (i = 0; i < idx; i++, page++) {
        __free_pages_bootmem(page, 0);
        count++;
    }
    total += count;
    bdata->node_bootmem_map = NULL;

    return total;
} /* End free_all_bootmem_core */
```

free_all_bootmem_core 函数清除 Boot Memory 的物理页面空间后,将释放物理页面的位图信息,即 bdata->node_bootmem_map 占用的物理页面空间。最后,将所有被释放的物理页面数存入 total 变量中返回。

7.3 Slab 分配器

7.2 节详细介绍了基于页面的物理内存管理策略 Buddy System,该策略是 Linux 系统内存管理的基础。但是 Linux 系统使用的内存大小往往不足一个页面,如果仍然使用 Buddy System 提供的操作函数进行页面申请,物理内存将不能被充分利用,采用这种做法将会在 Linux 系统中留下许多碎片,因为 Buddy Sytem 中内存的最小单位为一个物理页面,即使应用程序只需要申请 32 个字节大小的数据,Buddy System 也将提供一个 4 KB 的完整物理页面。为此 Linux 系统引入了 Slab 分配器(Slab Allocator)申请小块物理内存空间。

Slab 分配器首先在 Solaris 系统中使用。Linux 系统在 2.2 版本中,就开始使用这种物理内存管理策略。与其他的内存分配器相比,Slab 分配器有许多优点,如下所示:

- Slab 分配器将内存分解为一个个较小的数据对象(Object)进行管理,可以有效地提高内存利用率。
- 在 Slab 分配器中,一些在 Linux 系统中常用的数据结构,可以使用专用数据对象,而普通内存使用通用数据对象。
- Slab 分配器释放数据对象时,并不会随意释放这些数据对象,而是将这些数据对象暂存在 Slab 分配器的缓存中。当 Linux 系统重新申请这些数据对象时,Slab 分配器可以首先从这些缓冲中,获得数据对象。这些在数据缓冲中的数据对象,不需要重新进行初始化,从而大大提高了 Slab 分配器中数据对象的申请与释放效率。
- Slab 分配器还合理利用了处理器的 Cache。Slab 分配器尽量让相邻的数据对象使用不同的 Cache 行,从而有效避免了因为 Cache 颠簸而引发的数据利用率较低的问题。

Slab 分配器基于 Buddy System 系统,但是在多数情况下,Slab 分配器优先使用自身提供的缓冲区,尽量减少与 Buddy System 系统的交互。Slab 分配器,在物理内存中,维护了许多缓冲区。Linux 系统将这些缓冲区称为 Cache。提醒读者注意,Slab 分配器中的 Cache 与处理器内部的物理 Cache 没有任何关系。

Linux 系统又将这些 Cache 分割成为一个个 slab 容器,每一个 slab 可以由一个或者多个物理地址连续的页面组成,slab 由许多个数据对象(object)组成。

在 Linux 系统中,Slab 分配器的数据对象大小被预先定义为 2^5、2^6、...、2^{16}、2^{17}字节等。Linux 系统允许的数据大小在 ./include/linux/kmalloc_size.h 文件中定义。

Linux 系统规定,在 Slab 分配器中,最小的数据对象大小为一个物理 Cache 行长度,这样做的目的是为了加块内存的分配与释放速度。PowerPC E500 内核的 Cache 行长度为 2^5 字节,因此基于 E500 内核的 Linux PowerPC,其数据对象的大小为 2^5 的整数倍。读者可以使用"cat"命令查看/proc/slabinfo 文件,获得在当前系统中,Slab 分配器所管理的内存信息。Linux 系统使用双向链表将大小相同的 Cache 链接在一起。

Cache,slab 和 object 的逻辑关系如图 7-9 所示。

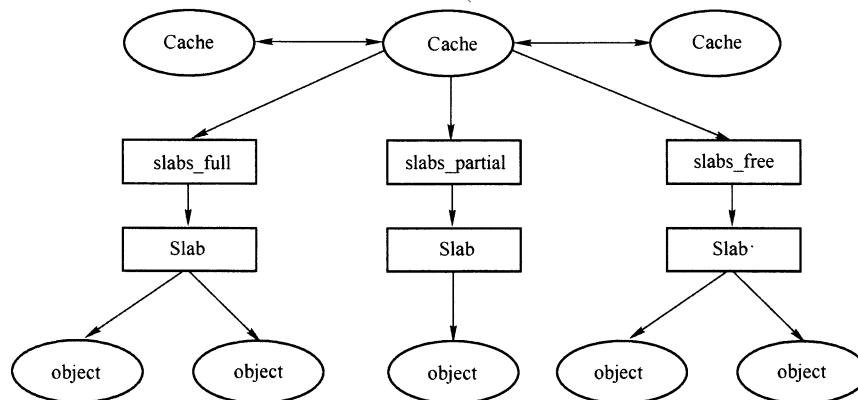


图 7-9 Cache 结构图

Slab 分配器包含两类 Cache,分别为通用 Cache 和专用 Cache。在 Linux 系统中,程序申请普通内存空间时,使用通用 Cache;申请 Linux 系统常用的一些数据结构空间时,将使用专用 Cache。在一个 Cache 中,包含三个 slab 队列,分别为 slabs_full、slabs_partial 和 slabs_free,这

些 slab 队列包含一系列 Slab 结构。其中每一个结构由若干个数据对象(Object)组成,Object 是 Slab 分配器进行内存分配的基本单位。

7.3.1 Slab 分配器的主要数据结构

在 Slab 分配器中,主要数据结构有 kmem_cache、cache_sizes 和 slb。

其中 kmem_cache 结构对 Cache 进行描述,该结构也被称为 Cache 描述符;cache_sizes 结构存放 Slab 分配器中的通用 Cache;而 slb 结构描述在 Cache 中的 slab。

1. kmem_cache 结构

kmem_cache 结构包含了 Cache 的全部描述信息,该结构的定义在 ./mm/slab.c 文件中,其主要数据成员的含义如下:

```
struct kmem_cache {
    struct array_cache *array[NR_CPUS];
```

array 参数优化 Linux 系统从 Cache 中获取和释放物理内存的速度。在实现数据结构 kmem_cache 时,Linux 系统考虑了许多硬件 Cache 的细节。现代处理器内核一定包含 L1 Cache,而芯片设计者可以根据具体应用的不同,选择不同大小的 L2 Cache。有的处理器在片内不包含 L2 Cache。

为此,Linux 系统仅根据 L1 Cache 行长度对程序进行优化。在 Linux 系统中,当前处理器的 Cache 行长度被保存在宏 L1_CACHE_BYTES 中。

在 Linux PowerPC 中,宏 L1_CACHE_BYTES 的定义在 ./include/asm-powerpc/cache.h 文件中,其值为 32B,即 256b。Linux 系统针对 L1 Cache 结构进行了许多优化,其主要的优化措施如下:

(1) 将一个结构中经常使用的数据成员放在该数据结构的开始处。在 kmem_cache 结构中,array 参数最常用。

(2) 尽量将数据成员按照 Cache 行对界。

(3) Linux SMP 系统极力避免将一块内存区的数据分散放入不同 CPU 的 L1 Cache 中。虽然使用 MESI 算法,可以解决 L1 Cache 的共享一致性的问题,但在并行算法中,L1 Cache 颠簸将极大影响整个系统的效率。

Linux 系统在设计数据结构 kmem_cache 时,首先将 array 参数放在最开始处,因为在每次申请和释放内存时,都需要使用 array 参数。在 Linux 系统中,array 参数作为 Cache 的缓存,保存刚刚被释放的内存空间。在 Slab 系统申请内存空间时,优先使用存放在 array 参数中的缓存;而释放内存空间时,将空间直接释放到 array 中。

对于 Linux SMP,array 数组一共有 NR_CPUS 个数据成员,目的是为了将 Linux SMP 系统的每个处理器申请的空间放入各自的 array 中,从而这些数据将会出现在各自处理器的 L1 Cache 中,以减少 L1 Cache 的数据颠簸。

```
unsigned int batchcount;
unsigned int limit;
unsigned int shared;
unsigned int buffer_size;
```

● batchount 参数表示当前 Cache 为空时,需要将多少个数据对象(object)加入到 Cache 中。为提高效率,Linux 系统一次将多个数据对象填充到 Cache 中。
● limit 参数保存当前 Cache 中存放的数据对象的最大个数,这个参数可调。
● shared 参数表示在当前 Cache 中,有多少数据对象是多个处理器共享的。
● buffer _ size 参数保存当前 Cache 数据对象的大小。

```
struct kmem_list3 *nodelists[MAX_NUMNODES];
```

nodelists 参数是指向 kmem _ list3 结构的数组指针。kmem _ list3 结构的定义在 ./mm/slab.c 文件中。在 nodelists 数组指针中,包含了 slabs _ full, slabs _ partial 与 slabs _ free 三种指针,以提高 Slab 分配器管理内存的效率。

其中 slabs _ full 指向没有空闲的数据对象链表,slabs _ partial 指向部分空闲、部分不空闲的数据对象链表,而 slabs _ free 指向空闲的数据对象链表。在一个 Cache 中,包含许多 Slab,在这些 Slab 中,又包含许多数据对象 Object。使用 Slab 分配器,进行内存申请时,将以数据对象 Object 为基本单位进行。

```
    unsigned int flags;   /* constant flags */
    unsigned int num;     /* # of objs per slab */
    unsigned int gfporder;
    gfp_t gfpflags;
    size_t colour;
    unsigned int colour_off;
    struct kmem_cache *slabp_cache;
    unsigned int slab_size;
    unsigned int dflags;
    void (*ctor)(void *, struct kmem_cache *, unsigned long);
    void (*dtor)(void *, struct kmem_cache *, unsigned long);
    const char *name;
    struct list_head next;
...
} /* End kmem_cache */
```

● flags 参数描述当前 Cache 的状态。
● num 参数描述在当前 Cache 的 Slab 中含有多少个数据对象。在同一个 Cache 中,数据对象的大小固定。
● gfporder 参数存放在一个 slab 中物理地址连续页面个数的对数(以 2 为底)。例如,在一个 slab 中,包含 8 个物理地址连续的页面,则该参数的值为 3。
● gfpflags 参数存放使用 Buddy System 进行页面申请时的状态信息,其含义与 alloc _ pages 函数的 gfp _ mask 相同。
● colour 参数表示在当前 Cache 中有多少种颜色。该参数用来优化 L1 Cache。
● colour _ off 参数确定 Slab 中数据对象之间的间隔。Slab 分配器设置 colour 参数的目的是保证在同一 Slab 中的数据对象,尽可能不出现在一个物理 Cache 行中。这一点保证了 Linux 系统可以有效利用硬件 L1 Cache。

- slabp _ cache 参数存放 Slab 描述符。在 Linux 系统中,Slab 描述符可以与 Slab 中的数据对象共享同一空间,也可以与数据对象的空间独立。如果 Slab 描述符的空间与数据对象空间独立,则 slabp _ cache 参数将指向 Slab 描述符,如果 Slab 描述符的空间与数据对象共享同一段空间,则 slabp _ cache 参数为 NULL。
- ctor 和 dtor 参数指向当前 Cache 的构造与析构函数。在 Slab 分配器中,使用了面向对象程序设计的一些技术。如果这两个参数不为空,Linux 系统在创建和释放数据对象时将自动执行这两个函数。
- name 参数保存当前 Cache 的名字。在 Linux 系统中,不同的 Cache 不能使用相同的名字,因此必须使用 name 参数加以区分。
- next 指针指向下一个 Cache 描述符。在 Linux 系统中,next 指针将所有 Cache 组成一个链表。

2. cache _ sizes 结构

cache _ sizes 结构的定义在 ./include/linux/slab _ def.h 文件中。该结构描述 Linux 系统中的通用 Cache,其源代码如下:

```
/* Size description struct for general caches. */
struct cache_sizes {
    size_t          cs_size;
    struct kmem_cache *cs_cachep;
    struct kmem_cache *cs_dmacachep;
};
```

- cs _ size 参数表示当前描述 Cache 的大小。在 Linux 系统中,Cache 的大小为 $2^5 \sim 2^{17}$,因此 cs _ size 的值在 $2^5 \sim 2^{17}$之间。
- cs _ cachep 和 cs _ dmacachep 参数指向 Cache 描述符。其中,cs _ cashep 参数描述存放在 ZONE _ NORMAL 数据区域的通用 Cache;cs _ dmacachep 参数描述存放在 ZONE _ DMA 数据区域的通用 Cache。

在 Linux 系统中,所有通用内存申请,如 kmalloc _ node、kmalloc 和 kzalloc 等函数申请内存时,都将使用通用 Cache。在 Linux 系统中,通用 Cache 最为常用。因此 Linux 系统定义了一个全局数组 malloc _ sizes,保存所有大小不同的通用 Cache,该全局数组的定义如下:

```
/*
 * These are the default caches for kmalloc. Custom caches can have other sizes.
 */
struct cache_sizes malloc_sizes[] = {
#define CACHE(x) { .cs_size = (x) },
#include <linux/kmalloc_sizes.h>
  CACHE(ULONG_MAX)
#undef CACHE
};
```

这段代码首先对宏 CACHE 重新定义,然后使用 kmalloc _ sizes.h 文件,初始化全局数组 malloc _ sizes 的所有 .cs _ size 参数。而 CACHE(ULONG _ MAX)存放 malloc _ sizes 数组的结

束标志。在 malloc _ sizes 数组中，静态分配了大小为 $2^5 \sim 2^{17}$B，一共 13 个 cache _ sizes 结构的数据成员，每一个数据成员包含两个指向 Cache 的指针，分别为 cs _ cachep 和 cs _ dmacachep。全局数组 malloc _ sizes 的结构如图 7-10 所示。

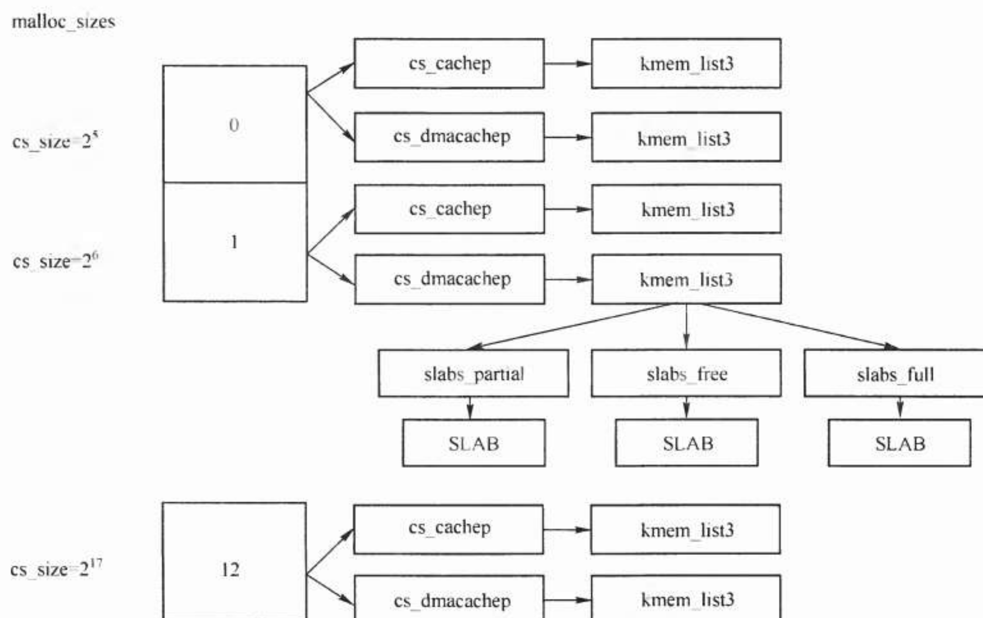


图 7-10 malloc _ sizes 数据结构图

在 Slab 分配器中，通用 Cache 根据其大小不同，存放在 malloc _ sizes 数组的相应位置中。例如数据对象的大小为 2^5B 的通用 Cache 存放在 malloc _ sizes[0]中；大小为 2^6B 的通用 Cache 存放在 malloc _ sizes[1]中，并以此类推。专用 Cache 并不存放在这个数组中，而是独立存放。

3. Slab 描述符

在 Linux 系统中，在一个 Cache 中包含许多个 Slab，slab 结构描述这些 Slab 的属性。slab 结构的定义在 ./mm/slab.c 文件中，该结构的详解如下：

```
struct slab {
    struct list_head list;
    unsigned long colouroff;
    void *s_mem;        /* including colour offset */
    unsigned int inuse;   /* num of objs active in slab */
    kmem_bufctl_t free;
    unsigned short nodeid;
};
```

- list _ head list 参数将 Slab 组成一个链表。由上文可知，在一个 Cache 中，共有三种 Slab 链表，分别是 slabs _ full，slabs _ partial 与 slabs _ free。
- colouroff 参数描述在当前 Slab 中第一个数据对象 object 在 s _ mem 中的偏移，其地址为 s _ mem + colouroff。

- s _ mem 参数是在当前 Slab 中第一个数据对象的基地址。
- inuse 参数描述在当前 Slab 中有多少个有效的数据对象。
- free 参数用来快速查找在当前 Slab 中的空闲数据对象，free 参数指向 kmem _ bufctl _ t 数组，本书将在 7.3.3 节中，详细介绍 kmem _ bufctl _ t 数组。
- nodeid 参数描述当前 Slab 的 ID 号。

7.3.2 Cache 的管理

Linux 系统以下使用几个全局变量，初始化 Slab 分配器中的 Cache。

(1) 全局变量 cache _ cache。cache _ cache 变量是一个 kmem _ cache 数据类型，cache _ cache 变量是 Linux 系统创建的第一个 Cache 描述符。

这个 Cache 描述符存放 Linux 系统其他的 Cache 描述符。在 Slab 分配器初始化时，不能使用 Slab 分配器提供的操作函数，如 kmem _ cache _ create 函数，创建 cache _ cache 变量。因为此时，Linux 系统并没有将 Slab 分配器初始化完毕。

Linux 系统必须静态初始化全局变量 cache _ cache。在 Linux 系统中，cache _ cache 属于专用 Cache 描述符，该 Cache 描述符是 Slab 分配器创建其他 Cache 描述符的基础，也是整个 Slab 分配器的基础变量。该变量 cache _ cache 在 ./mm/slab.c 文件中定义，其源代码如下：

```
/* internal cache of cache description objs */
static struct kmem_cache cache_cache = {
    .batchcount = 1,
    .limit = BOOT_CPUCACHE_ENTRIES,
    .shared = 1,
    .buffer_size = sizeof(struct kmem_cache),
    .name = "kmem_cache",
};
```

(2) 全局变量 cache _ chain。在 Linux 系统中，所有 Cache 描述符组成一个双向队列，全局变量 cache _ chain 保存该队列的首指针。在 Linux 系统中，全局变量 cache _ chain 指向的第一个 Cache 描述符为 cache _ cache，因为在 Linux 系统中，cache _ cache 描述符是第一个 Cache 描述符。

(3) 全局变量 cache _ chain _ sem。该变量保存访问 cache _ chain 链表时的信号量。

(4) 全局数组 malloc _ sizes。该变量在上文中曾详细介绍。

Slab 分配器的初始化使用了以上这些全局变量。Slab 分配器的初始化由以下主要内容组成。

- 初始化全局变量 cache _ chain，cache _ cache。这些变量由 Linux 系统静态初始化。
- 初始化通用 Cache 描述符。通用 Cache 描述符主要存放处理器中的通用内存，所有的通用 Cache 描述符将根据数据对象的大小，存放在相应的 malloc _ sizes 数组中。Linux 系统调用 kmem _ cache _ init 函数，初始化通用 Cache 描述符。
- 初始化专用 Cache 描述符。在 Linux 系统中，专用 Cache 描述符存放一些常用的数据结构，如 signal，inode，mm _ struct 结构等。在 Linux 设备驱动程序中，可以建立专用 Cache 描述符，将一些使用频率较高的结构，存放在专用 Cache 描述符中，以提高设备驱

动程序的效率。与通用 Cache 描述符不同，专用 Cache 描述符不加入到 malloc_sizes 数组中，而是使用专用 Cache 指针维护。Linux 系统使用 kmem_cache_create 函数创建专用 Cache 描述符。

1. 通用 Cache 的创建

Linux 系统使用 kmem_cache_init 函数，初始化 Slab 分配器和所有通用 Cache。在 Linux 系统初始化时，该函数被 start_kernel 函数调用，其源代码在 ./mm/slab.c 文件中。该函数没有输入参数也没有返回值，源代码详解如下：

```
/* kmem_cache_init 函数源代码片段 1 */
void __init kmem_cache_init(void)
{
  size_t left_over;
  struct cache_sizes *sizes;
...
  for (i = 0; i < NUM_INIT_LISTS; i++) {
    kmem_list3_init(&initkmem_list3[i]);
    if (i < MAX_NUMNODES)
      cache_cache.nodelists[i] = NULL;
  }
  if (num_physpages > (32 << 20) >> PAGE_SHIFT)
    slab_break_gfp_order = BREAK_GFP_ORDER_HI;
```

kmem_cache_init 函数在 Buddy System 系统初始化函数 mem_init 函数之后执行。由此可见，在 Linux 系统中，Slab 分配器的初始化在 Buddy System 的初始化之后，因此在 kmem_cache_init 函数中，可以使用 Buddy System 中提供的物理页面的申请与释放函数。

在这段代码中，首先将全局变量 initkmem_list3 初始化，这个全局变量在初始化全局变量 cache_cache 中使用。随后这段代码将 cache_cache 变量的 nodelists 参数置为空。在 UMA 结构中，MAX_NUMNODES 为 1。

全局变量 num_physpages 保存在当前存储器系统中物理页面的数量。当全局变量 num_physpages 的值大于 8 KB(对于 32 位处理器，即当前存储器的内存大于 32 MB)，全局变量 slab_break_gfp_order 将被置为 BREAK_GFP_ORDER_HI，否则该全局变量的值为 BREAK_GFP_ORDER_LO。全局变量 slab_break_gfp_order 在 kmem_cache_create 函数中使用，本书将在下文详细介绍 kmem_cache_create 函数。

```
/* kmem_cache_init 函数源代码片段 2 */
...
   /* 1) create the cache_cache */
   INIT_LIST_HEAD(&cache_chain);
   list_add(&cache_cache.next, &cache_chain);
   cache_cache.colour_off = cache_line_size();
   cache_cache.array[smp_processor_id()] = &initarray_cache.cache;
   cache_cache.nodelists[node] = &initkmem_list3[CACHE_CACHE];
```

```
cache_cache.buffer_size = ALIGN(cache_cache.buffer_size,
        cache_line_size());
cache_cache.reciprocal_buffer_size =
  reciprocal_value(cache_cache.buffer_size);
```

这段程序的执行流程如下：

(1) 创建全局变量 cache_chain，并将 cache_cache 变量加入到 cache_chain 链表中。

(2) 初始化全局变量 cache_cache，将 colour_off 的值初始化为一个硬件 Cache 行长度，以尽可能保证相邻的数据对象不会在同一个 Cache 行中命中。

(3) 使用 initarray_cache.cache 初始化 cache_cache 的 array 参数。

(4) 使用 initkmem_list3 数组初始化 cache_cache 的 nodelists 参数。initarray_cache 变量和 initkmem_list3 数组在 ./mm/slab.c 文件中，这两个变量被静态初始化。

(5) 最后将全局变量 cache_cache 的 buffer_size 进行 Cache 行对齐，buffer_size 的值在 Linux 系统加载时置为 sizeof(struct kmem_cache)。

```
/* kmem_cache_init 函数源代码片段 3 */
  for (order = 0; order < MAX_ORDER; order++) {
    cache_estimate(order, cache_cache.buffer_size,
      cache_line_size(), 0, &left_over, &cache_cache.num);
    if (cache_cache.num)
      break;
  }
```

此段代码使用 cache_estimate 函数，依次从 Buddy System 物理地址连续的页面中估算是否有存放 cache_cache 所需要的数据空间，并计算在这个物理地址连续的页面中(可能是 1 个物理地址连续的页面，也可能是多个物理地址连续的页面)，可以存放多少个数据对象，并将此结果存放在 cache_cache.num 参数中。

同时此函数将剩余空间的大小存到 left_over 变量中。在 Buddy System 中，内存分配的基本单位是一个物理页面，因此使用这个物理地址连续的页面分配完 cache_cache 变量所需要的空间后，将剩余一段空间。

```
/* kmem_cache_init 函数源代码片段 4 */
    BUG_ON(!cache_cache.num);
    cache_cache.gfporder = order;
    cache_cache.colour = left_over / cache_cache.colour_off;
    cache_cache.slab_size = ALIGN(cache_cache.num * sizeof(kmem_bufctl_t) +
            sizeof(struct slab), cache_line_size());
```

这段程序完成全局变量 cache_cache 的最终初始化。此时有关全局变量 cache_cache 的初始化完全结束。此后 Linux 系统可以使用 cache_cache 全局变量，通过 kmem_cache_create 函数创建其他 Cache 描述符。

```
/* kmem_cache_init 函数源代码片段 5 */
  /* 2+3) create the kmalloc caches */
```

```
    sizes = malloc_sizes;
    names = cache_names;

    /*
     * Initialize the caches that provide memory for the array cache and the
     * kmem_list3 structures first. Without this, further allocations will
     * bug.
    */
    sizes[INDEX_AC].cs_cachep = kmem_cache_create(names[INDEX_AC].name,
            sizes[INDEX_AC].cs_size,
            ARCH_KMALLOC_MINALIGN,
            ARCH_KMALLOC_FLAGS|SLAB_PANIC,
            NULL, NULL);
```

这段程序首先调用 kmem_cache_create 函数，创建用于存放“arraycache_init 结构所需要的物理空间”的通用 Cache 描述符，然后根据“arraycache_init 结构的大小”将这个通用 Cache 描述符存放入全局数组 malloc_sizes 相应的队列中。Cache 描述符的 array 参数，作为 Cache 描述符的缓存，使用 arraycache_init 结构存放。

在这个通用 Cache 描述符创建完成之后，Linux 系统可以使用 Slab 分配器所提供的函数为其他 Cache 描述符的 array 参数在 Slab 分配器中，申请内存空间。

```
/* kmem_cache_init 函数源代码片段 6 */
    if (INDEX_AC != INDEX_L3) {
      sizes[INDEX_L3].cs_cachep =
        kmem_cache_create(names[INDEX_L3].name,
          sizes[INDEX_L3].cs_size,
          ARCH_KMALLOC_MINALIGN,
          ARCH_KMALLOC_FLAGS|SLAB_PANIC,
          NULL, NULL);
    }
```

如果 kmem_list3 结构和 arraycache_init 结构使用同一个通用 Cache 描述符，不执行 if 语句；否则，这段程序进一步创建用于存放“kmem_list3 结构所需要的物理空间”的通用 Cache 描述符。

在 Slab 系统的初始化中，源代码片段 5 和 6 比较费解，这两段代码涉及了一个“先有鸡还是现有蛋”的互锁问题。kmem_cache_create 函数需要使用 kmalloc 函数，分配 Cache 描述符的 array 和 kmem_list3 结构使用的内存空间，而 kmalloc 函数需要在通用 Cache 描述符建立之后才能够被使用。

为此，Linux 系统调用 kmem_cache_create 函数创建 array 和 kmem_list 结构，以及创建通用 Cache 描述符的函数执行路径，与创建专用 Cache 描述符的执行路径并不相同。说得具体些，就是 kmem_cache_create->setup_cpu_cache 函数的执行路径不同。

setup_cpu_cache 函数设置了一个全局枚举变量 g_cpucache_up。Linux 系统使用了这

个全局变量,合理解决了在 SLAB 分配器初始化时,存在的"先有鸡还是现有蛋"互锁问题。g_cpucache_up 变量定义如下,其初始值为 NONE。

```
static enum {
    NONE,
    PARTIAL_AC,
    PARTIAL_L3,
    FULL
} g_cpucache_up;
```

下文结合 setup_cpu_cache 函数的源代码说明 g_cpucache_up 的用途。setup_cpu_cache 函数的源代码详解如下:

```
static int setup_cpu_cache(struct kmem_cache *cachep)
{
    if (g_cpucache_up == FULL)
      return enable_cpucache(cachep);
```

在 setup_cpu_cache 函数中,如果 g_cpucache_up 变量为 FULL,则执行 enable_cpucache 函数,该函数的简要说明见下文。在 Linux 系统的初始化过程中,array 和 kmem_list 结构使用的通用 Cache 描述符和其他通用 Cache 描述符创建完毕后,g_cpucache_up 被设置为 FULL。由此可见 Linux 系统创建专用 Cache 描述符时,一定使用 enable_cpucache 函数。

```
    if (g_cpucache_up == NONE) {

      cachep->array[smp_processor_id()] = &initarray_generic.cache;

      set_up_list3s(cachep, SIZE_AC);
      if (INDEX_AC == INDEX_L3)
        g_cpucache_up = PARTIAL_L3;
      else
        g_cpucache_up = PARTIAL_AC;
    } else {
      cachep->array[smp_processor_id()] =
        kmalloc(sizeof(struct arraycache_init), GFP_KERNEL);

      if (g_cpucache_up == PARTIAL_AC) {
        set_up_list3s(cachep, SIZE_L3);
        g_cpucache_up = PARTIAL_L3;
      } else {
        int node;
        for_each_online_node(node) {
          cachep->nodelists[node] =
            kmalloc_node(sizeof(struct kmem_list3),
```

```
                GFP_KERNEL, node);
            BUG_ON(!cachep->nodelists[node]);
            kmem_list3_init(cachep->nodelists[node]);
        }
    }
}
...
} /* End setup_cpu_cache */
```

该函数的主体是当 g_cpucache_up 不为 FULL 时的情况，这段程序需要针对“条件(g_cpucache_up == NONE)为真或者为假”两种情况，分别进行讨论。

(1) 条件(g_cpucache_up == NONE)为真，表示在 Linux 系统中，还没有创建通用 Cache 描述符，此时执行这段代码一定是为了创建存放 arraycache_init 结构的通用 Cache 描述符，arraycache_init 结构使用的通用 Cache 描述符是 Linux 系统创建的第一个 Cache 描述符。这段程序执行流程如下：

- 在创建存放 arraycache_init 结构的 Cache 描述符时，g_cpucache_up 一定为初始值 None。此时 Linux 系统使用静态变量 initarray_generic.cache，初始化该 Cache 描述符的 arraycache_init 结构，而不是使用 kmalloc 函数。在 Slab 分配器没有初始化完成时，无法使用 kmalloc 函数。
- 调用 set_up_list3s 函数，初始化该 Cache 描述符的 kmem_list 结构。set_up_list3s 函数使用全局数组 initkmem_list3，初始化 Cache 描述符的 kmem_list 结构，不能使用 kmalloc 函数。
- 如果 INDEX_AC 等于 INDEX_L3，即存放 arraycache_init 结构和存放 kmem_list 使用相同的通用 Cache 描述符，此时将 g_cpucache_up 赋值为 PARTIAL_L3，否则将 g_cpucache_up 赋值为 PARTIAL_AC。

(2) 条件(g_cpucache_up == NONE)为假，表示 arraycache_init 结构使用的通用 Cache 描述符已经创建完毕，这段代码用来创建除存放 arraycache_init 结构的通用 Cache 描述符之外的所有通用 Cache 描述符。这段程序执行流程如下：

- 使用 kmalloc 函数，创建当前 Cache 描述符的 arraycache_init 结构。此时由于存放 arraycache_init 结构的通用 Cache 描述符已经创建完毕，因此可以使用 kmalloc 函数在 SLAB 分配器中，申请内存空间。但是 kmalloc 函数只能申请 arraycache_init 结构使用的内存空间，因为此时 Linux 系统并没有创建其他通用 Cache 描述符。
- 如果 g_cpucache_up 等于 PARTIAL_AC，表示存放 arraycache_init 结构和存放 kmem_list 并不使用相同的通用 Cache 描述符。因此需要调用 set_up_list3s 函数初始化该 Cache 描述符的 kmem_list 结构，并将 g_cpucache_up 赋值为 PARTIAL_L3。
- 如果 g_cpucache_up 等于 PARTIAL_AC，表示存放 kmem_list 结构的通用 Cache 描述符已经创建完毕。此时这段程序将调用 kmalloc_node 函数(该函数与 kmalloc 函数类似，都用来申请内存空间)分配该 Cache 描述符中 kmem_list 结构的内存空间。之后这段程序调用 kmem_list3_init 初始化 kmem_list 结构。

这部分程序是 SLAB 分配器的难点，读者需要了解 kmem_cache_create 和 kamlloc 函数

之后，才可能真正理解这段源代码。如果读者读到此处已经完全理解了上述代码的全部含义，那么可以略过 7.3 节。

kmem_cache_init 函数创建完毕用于存放 arraycache_init 结构所需要的物理空间的通用 Cache 描述符和存放 kmem_list3 结构所需要的物理空间的通用 Cache 描述符后，将执行以下程序：

```
/* kmem_cache_init 函数源代码片段 7 */
 slab_early_init = 0;

 while (sizes->cs_size != ULONG_MAX) {
  /*
   * For performance, all the general caches are L1 aligned.
   * This should be particularly beneficial on SMP boxes, as it
   * eliminates "false sharing".
   * Note for systems short on memory removing the alignment will
   * allow tighter packing of the smaller caches.
   */
  if (!sizes->cs_cachep) {
    sizes->cs_cachep = kmem_cache_create(names->name,
         sizes->cs_size,
         ARCH_KMALLOC_MINALIGN,
         ARCH_KMALLOC_FLAGS|SLAB_PANIC,
         NULL, NULL);
  }

  sizes->cs_dmacachep = kmem_cache_create(names->name_dma,
       sizes->cs_size,
       ARCH_KMALLOC_MINALIGN,
       ARCH_KMALLOC_FLAGS|SLAB_CACHE_DMA|
         SLAB_PANIC,
       NULL, NULL);
  sizes++;
  names++;
 }
```

上述代码将全局数组 malloc_sizes 中的其他数据成员进行初始化，即创建其他剩余的通用 Cache 描述符。注意此时，存放 array 结构和 kmem_list3 结构所需要的物理空间的通用 Cache 描述符已经被创建完毕，因此在这段程序需要判断 sizes->cs_cachep 是否为空。这段程序的执行流程如下：

- 当 malloc_sizes 的 cs_size 参数为 ULONG_MAX 时，表示当前程序已经完成对 malloc_sizes 数组的扫描。
- 这段代码将为 malloc_sizes 中所有数据成员的 cs_cachep 链表和 cs_dmacachep 链表空间建立对应的 Cache 描述符，即所有通用 Cache 描述符。

至此,Slab 分配器的初始化已经基本完毕,在 Linux 系统中,所有通用 Cache 描述符初始化完毕。此时我们可以使用 Slab 分配器提供的 kmalloc 函数和 kfree 函数,进行物理内存空间的申请与释放。

```
/* kmem_cache_init 函数源代码片段 8 */
  /* 4) Replace the bootstrap head arrays */
  {
     struct array_cache *ptr;

     ptr = kmalloc(sizeof(struct arraycache_init), GFP_KERNEL);

     local_irq_disable();
     BUG_ON(cpu_cache_get(&cache_cache) != &initarray_cache.cache);
     memcpy(ptr, cpu_cache_get(&cache_cache),
          sizeof(struct arraycache_init));
     /*
      * Do not assume that spinlocks can be initialized via memcpy:
      */
      spin_lock_init(&ptr->lock);

      cache_cache.array[smp_processor_id()] = ptr;
      local_irq_enable();

      ptr = kmalloc(sizeof(struct arraycache_init), GFP_KERNEL);

      local_irq_disable();
      BUG_ON(cpu_cache_get(malloc_sizes[INDEX_AC].cs_cachep)
         != &initarray_generic.cache);

      memcpy(ptr, cpu_cache_get(malloc_sizes[INDEX_AC].cs_cachep),
         sizeof(struct arraycache_init));
      /*
       * Do not assume that spinlocks can be initialized via memcpy:
       */
       spin_lock_init(&ptr->lock);

       malloc_sizes[INDEX_AC].cs_cachep->array[smp_processor_id()] =
          ptr;
       local_irq_enable();
  }
```

这是 kmem_cache_init 函数中另一段较难理解的代码。其详细说明如下。

● 首先这段程序使用 kmalloc 函数,在通用 Cache 描述符中,申请一段与 arraycache_init

结构大小相同的内存空间。

- 将 cache _ cache->array 中的数据，即 initarray _ cache.cache 中的数据复制到 ptr 中。
- 使 cache _ cache->array 指向 ptr 指针，之后全局变量 initarray _ cache.cache 不会再被使用。当程序员使用 kmem _ cache _ create 函数创建 Cache 描述符时，将使用这个 ptr 指针。至此全局变量 initarray _ cache 已经完成了 Boot Strap 的任务。
- 最后，这段程序使用 kmalloc 函数，重新分配一段与 arraycache _ init 结构大小相同的内存空间，然后将存放在 initarray _ generic.cache 中的数据复制到 ptr 中。最后将存放"array 结构结构所需要的物理空间"的通用 Cache 描述符 cs _ cachep->array 参数赋值为 ptr，至此全局变量 initarray _ generic 已经完成了 Boot Strap 的任务，将不再被使用。

```
/* kmem_cache_init 函数源代码片段 9 */
  /* 5) Replace the bootstrap kmem_list3's */
  {
    int nid;

    /* Replace the static kmem_list3 structures for the boot cpu */
    init_list(&cache_cache, &initkmem_list3[CACHE_CACHE], node);

    for_each_online_node(nid) {
      init_list(malloc_sizes[INDEX_AC].cs_cachep,
          &initkmem_list3[SIZE_AC + nid], nid);

      if (INDEX_AC != INDEX_L3) {
          init_list(malloc_sizes[INDEX_L3].cs_cachep,
              &initkmem_list3[SIZE_L3 + nid], nid);
      }
    }
  }
```

这段程序与源代码片段 8 的执行原理类似，其主要作用是结束 kmem _ list3 数组的 Boot Strap 使命。读者可以自行阅读这段代码作为练习。

```
/* kmem_cache_init 函数源代码片段 10 */
  /* 6) resize the head arrays to their final sizes */
  {
    struct kmem_cache *cachep;
    mutex_lock(&cache_chain_mutex);
    list_for_each_entry(cachep, &cache_chain, next)
      if (enable_cpucache(cachep))
        BUG();
    mutex_unlock(&cache_chain_mutex);
  }
```

这段函数使用 Slab 分配器中提供的标准内存申请函数，kmalloc _ node 函数，替换所有

Slab 分配器初始化时，分配的 Cache 描述符的 array 参数和 kmem _ list3 参数，并调整 array 参数和 kmem _ list3 参数使用的实际物理空间。这段函数的实现要点在于 enable _ cpucache 函数。enable _ cpucache 函数调用 alloc _ arraycache 和 alloc _ kmemlist 函数，分配 Cache 描述符中的 array 参数和 kmem _ list3 参数使用的空间。alloc _ arraycache 和 alloc _ kmemlist 函数会调用 kmalloc _ node 函数申请内存空间。

```
/* kmem_cache_init 函数源代码片段 11 */
    /* Annotate slab for lockdep -- annotate the malloc caches */
    init_lock_keys();

    /* Done! */
    g_cpucache_up = FULL;

    /*
     * Register a cpu startup notifier callback that initializes
     * cpu_cache_get for all new cpus
     */
    register_cpu_notifier(&cpucache_notifier);

    /*
     * The reap timers are started later, with a module init call: That part
     * of the kernel is not yet operational.
     */
}
```

最后这段程序将枚举变量 g _ cpucache _ up 设置为 FULL。至此 Slab 分配器的初始化全部完成，所有通用 Cache 已经被初始化完毕。此后，Linux 系统可以使用 Slab 分配器提供的函数，进行内存的分配与释放。

2. 专用 Cache 描述符的创建

由上文可知，kmem _ cache _ init 函数创建了所有通用 Cache，并将这些 Cache 描述符加入到全局链表 malloc _ sizes 和 cache _ chain 中。之后，Linux 系统可以使用一些通用内存申请与释放函数，获取与释放在通用 Cache 中的数据对象。

但是有时 Linux 系统为了提高对一些关键数据结构的访问速度，采用专用 Cache 保存这些数据结构。专用 Cache 符也将加入到全局链表 cache _ chain 中。Linux 系统的专用 Cache 共分为以下几类：

(1) 与文件系统相关的专用 Cache，这些 Cache 用来保存 dentry、names、inode、nfs 和 jffs2 _ inode 等结构。

(2) 与进程调度相关的专用 Cache，这些 Cache 用来保存 task _ struct、rpc _ tasks 等结构。

(3) 与内存管理相关的专用 Cache，这些 Cache 用来保存 vm _ area、mm _ struct 等结构。实际上，cache _ cache 也是专用 Cache 的一种，只是这个 Cache 的描述符需要手工创建，而不能使用 kmem _ cache _ create 函数。

(4) 与网络协议栈相关的专用 Cache，这些 Cache 用来保存 skbuff _ head、skbuff _ fclone 等

结构。

(5) 其他所有为程序优化创建的自定义专用 Cache 描述符。

在 Linux 系统中,有几百种专用 Cache,其定义各不相同。目前也并没有一个合理的规范对此进行命名,这种做法非常容易造成专用 Cache 描述符的名字污染。也许以后 Linux 系统会出现针对专用 Cache 描述符的命名规则,只是希望这种命名规范不要过于复杂,也不要有太多层次。在 Linux 系统中,通用描述符和专用描述符都是用 kmem_cache_create 函数创建的。

下文将以专用 Cache,task_struct_cachep 的创建为例,说明 kmem_cache_create 函数的实现过程。task_struct_cachep 在 fork_init 函数中创建。

fork_init 函数的源代码在 ./kernel/fork.c 文件中。在 Linux 系统初始化时,该函数被 start_kernel 函数调用。Linux 系统使用 alloc_task_struct 和 free_task_struct 函数通过专用 Cache,task_struct_cachep,创建和释放 task_struct 结构。

```
# define alloc_task_struct() kmem_cache_alloc(task_struct_cachep, GFP_KERNEL)
# define free_task_struct(tsk) kmem_cache_free(task_struct_cachep, (tsk))
static kmem_cache_t * task_struct_cachep;
...
void __init fork_init(unsigned long mempages)
{
...
  /* create a slab on which task_structs can be allocated */
  task_struct_cachep =
    kmem_cache_create("task_struct", sizeof(struct task_struct),
      ARCH_MIN_TASKALIGN, SLAB_PANIC, NULL, NULL);
...
}
```

task_struct_cachep 是静态全局变量,其作用范围在 fork.c 文件中。task_struct_cachep 是一个 kmem_cache_t 结构类型的指针,指向 kmem_cache_create 函数创建的专用 Cache 描述符。kmem_cache_create 函数的源代码在 ./mm/slab.c 文件中。kmem_cache_create 函数一共有 6 个输入参数,该函数执行成功后,返回已分配的专用 Cache 描述符指针。

```
struct kmem_cache *
kmem_cache_create (const char * name, size_t size, size_t align,
  unsigned long flags,
  void ( * ctor)(void *, struct kmem_cache *, unsigned long),
  void ( * dtor)(void *, struct kmem_cache *, unsigned long))
{
```

kmem_cache_create 函数的参数如下:

- name 参数。这个参数存放在/proc/slabinfo 文件中相应 Cache 描述符的名字。对于专用 Cache 描述符 task_struct_cachep,该值为"task_struct"。
- size 参数。这个参数存放在当前 Cache 描述符中,数据对象的大小。对于专用 Cache 描

述符 task _ struct，这个值为 sizeof(struct sask _ struct)。

- align 参数。这个参数描述当前 Cache 描述符中数据对象的对界单位。对于专用 Cache 描述符 task _ struct，这个值为 ARCH _ MIN _ TASKALIGN，即 L1 _ CACHE _ BYTES。
- flags 参数表示创建当前 Cache 描述符时的约束条件。对于专用 Cache 描述符 task _ struct _ cachep，这个值为 SLAB _ PANIC，即如果 kmem _ cache _ create 函数创建 Cache 描述符失败，将引发 panic 事件。
- ctor 和 dtor 参数。ctor 参数在 Slab 分配器创建数据对象时调用，而 dtor 参数在 Slab 系统释放数据对象时调用。

```
    size_t left_over, slab_size, ralign;
    struct kmem_cache *cachep = NULL, *pc;

    /*
     * Sanity checks... these are all serious usage bugs.
     */
    if (!name || in_interrupt() || (size < BYTES_PER_WORD) ||
        (size > (1 << MAX_OBJ_ORDER) * PAGE_SIZE) || (dtor && !ctor)) {
        printk(KERN_ERR "%s: Early error in slab %s\n", __FUNCTION__,
            name);
        BUG();
    }
...
    /* disable debug if necessary */
    if (ralign > BYTES_PER_WORD)
       flags &= ~(SLAB_RED_ZONE | SLAB_STORE_USER);
    /*
     * 4) Store it.
     */
```

这段程序首先对 kmem _ cache _ create 函数进行参数检查。随后对 cache _ chain 链表进行扫描，检查在 cache _ chain 链表中，是否已经有此类专用 Cache 描述符。在 cache _ chain 链表中，不允许 Cache 描述符同名。

```
    align = ralign;

    /* Get cache's description obj. */
    cachep = kmem_cache_zalloc(&cache_cache, GFP_KERNEL);
    if (!cachep)
      goto oops;
```

这段函数使用 kmem _ cache _ zalloc 函数，从 cache _ cache 描述符中，获得专用 Cache 描述符 task _ struct _ cachep 所使用的内存对象。由上文得知，cache _ cache 描述符存放所有 Cache 描述符使用的内存空间。

kmem _ cache _ zalloc 函数将依次调用__ cache _ alloc 函数、____ cache _ alloc 函数和 cpu _

cache _ get 函数。在这几个函数中，cpu _ cache _ get 函数最为重要。

cpu _ cache _ get 函数首先从 cache _ cache 的 array _ cache 参数中，获得所需要的 Cache 描述符；如果 cache _ cache->array _ cache 参数中没有数据对象，则使用 cache _ alloc _ refill 函数将 array _ cache 重新装填，之后获得相应的 Cache 描述符使用的内存空间。

```
/*
 * Determine if the slab management is 'on' or 'off' slab.
 * (bootstrapping cannot cope with offslab caches so don't do
 * it too early on.)
 */
if ((size >= (PAGE_SIZE >> 3)) && !slab_early_init)
  /*
   * Size is large, assume best to place the slab management obj
   * off-slab (should allow better packing of objs).
   */
  flags |= CFLGS_OFF_SLAB;
```

这段代码决定是否为 flags 参数加上 CFLGS _ OFF _ SLAB 标志，该标识决定当前 Cache 的 Slab 描述符是与数据对象共享内存空间，还是单独存放。

当 Cache 描述符中的数据对象所占空间大于 1/8 个物理页面，而且 slab _ early _ init 参数不为 1 时，当前 Cache 的 flag 参数将被加上 CFLGS _ OFF _ SLAB 标志，此时当前 Cache 的 Slab 描述将不与数据对象共享内存空间。这样做的主要目的是为了提高 Slab 描述符的空间利用率，以减少数据碎片。

当 Cache 中的数据对象大于 Slab 描述符所占的空间时，如果 Slab 描述符仍与数据对象共享内存空间，会造成一些空间浪费，所浪费的空间是 Slab 描述符的大小减当前 Cache 中数据对象的大小。当前 Cache 中的数据对象越大，Slab 分配器所浪费的空间就越大，1/8 个物理页面就是为此设置的一个阈值。

slab _ early _ init 参数为 kmem _ cache _ init 函数而设。在 kmem _ cache _ init 函数中，全局指针 cache _ cache 还没有被初始化好，此时调用 kmem _ cache _ create 函数，只能使 Slab 描述符与数据对象共享物理内存空间。

```
size = ALIGN(size, align);
left_over = calculate_slab_order(cachep, size, align, flags);
```

calculate _ slab _ order 函数的主要作用是，计算当前 Cache 的 Slab 中，可以存放多少个数据对象，在当前的 Cache 的 Slab 中还将剩余多少空间，并将这些参数分别存入到 Cache 描述符的 num 参数和 left _ over 参数中。Slab 使用 Buddy System 中物理地址页面连续的页面空间，calculate _ slab _ order 函数的执行流程如下所示：

(1) 使 gfp _ order = 0。

(2) 使用 cache _ estimate 函数，判断在 Buddy System 的 Zone→ free _ area[gfp _ order]中，是否有合适的空间分配给当前 Cache 描述符。如果有，获得临时的 num 和 remainder 参数，并继续执行；否则 gfp _ order++，转至第 1 步。

(3) 检查当前 Cache 的 Slab 描述符是否与数据对象共享内存空间，如果不是，则继续，否

则进一步判断，获得的临时 num 参数是否大于 slab 中容纳数据对象的阈值。如果 num 参数大于此阈值，则退出 calculate_slab_order 函数，并丢弃在第 2 步中获得临时的 num 和 reminder 参数。当然在 gfp_order 为 0 时，不可能出现 num 参数大于此阈值的情况，具体原因，留给读者分析。

(4) 将第 2 步获得的 num 和 reminder 参数，存入 Cache 描述符的 num 参数和 left_over 变量中。并将 gfp_order 变量存入 Cache 描述符的 gfp_order 参数中。

(5) 检查 flags 参数是否有 SLAB_RECLAIM_ACCOUNT，SLAB_RECLAIM_ACCOUNT 标识。这些标志用来创建与文件系统有关的 Cache 描述符，如 inode，romfs_inode，nfs_inode 等。这些 Cache 描述符最好只使用一个物理页面。当 SLAB_RECLAIM_ACCOUNT 标志有效时，将退出 calculate_slab_order 函数。

(6) 判断当前 Buddy System 中，物理连续的页面是否大于或者等于 slab_break_gfp_order。如果是，则直接退出 calculate_slab_order 函数，因为 Linux 的 Slab 分配器不鼓励使用比较大的物理地址连续的页面。在 Linux PowerPC 中，slab_break_gfp_order 的值为 1，表示如果当前 Cache 描述符使用的物理连续的页面大于 2^1 个，则退出 calculate_slab_order 函数，并且不做碎片检查。

(7) 判断使用当前物理连续的页面空间，其整个空间的浪费率是否小于等于 25%。如果结果为真，则退出 calculate_slab_order 函数，否则 gfp_order++，转至第 1 步进一步进行检查。

(8) 将 left_over 的值返回。

```
...
    cachep->colour_off = cache_line_size();
    /* Offset must be a multiple of the alignment. */
    if (cachep->colour_off < align)
      cachep->colour_off = align;
    cachep->colour = left_over / cachep->colour_off;
    cachep->slab_size = slab_size;
    cachep->flags = flags;
    cachep->gfpflags = 0;
    if (flags & SLAB_CACHE_DMA)
      cachep->gfpflags |= GFP_DMA;
    cachep->buffer_size = size;
    cachep->reciprocal_buffer_size = reciprocal_value(size);
```

这段程序的执行流程如下：

(1) 首先将当前 Cache 描述符的 colour_off 参数置为 L1_CACHE_BYTES，如果此值小于 kmem_cache_create 函数要求的对界大小 align，则将 colour_off 参数置为 align。

(2) colour_off 参数用来计算，在当前 Cache 描述符中，每个 Slab 之间的间隔。在 Cache 描述符中，有许多个 Slab，这些 Slab 中包含若干个数据对象。colour_off 参数用来确定在 Slab 中，每一个数据对象的偏移。

(3) 计算当前 Cache 描述符的 colour 参数，这个参数为 left_over(当前 Slab 的剩余空间)

除以 colour_off。left_over 和 colour_off 参数用来提高 Slab 分配器使用硬件 Cache 的效率。在现代处理器中，Cache 以一个个 Cache 行组织起来，其中每一个 Cache 行可以映射到几个指定的内存中。在 Slab 分配器中，设置 colour_off 和 colour 参数的主要目的是避免 Slab 中，相邻的数据对象使用同一个 Cache 行，以避免 Cache 颠簸。下文以一个实例说明 Slab 分配器如何使用 colour_off 和 colour 参数以避免 Cache 颠簸。

假定 left_over 为 140B，align 为 32B。此时 colour 参数为 140/32=4(需要取整)。为简便起见，我们假定在当前 Slab 中，第一个数据对象的地址偏移为 0，则第二个数据对象的地址偏移为 32，第三个为 64，第四个为 96，第五个为 128，第六个回归 0。采用这种办法将有效避免相邻的数据对象占用同一个物理 Cache 行。

(4) 初始化 cachep 的 flags，gfpflags 和 buffer_size 参数。

```
if (flags & CFLGS_OFF_SLAB) {
  cachep->slabp_cache = kmem_find_general_cachep(slab_size, 0u);
  BUG_ON(!cachep->slabp_cache);
}
cachep->ctor = ctor;
cachep->dtor = dtor;
cachep->name = name;
```

如果 Slab 描述符不与数据对象共享内存空间，则使用 kmem_find_general_cachep 函数，为 Slab 描述符找到合适的 Cache 描述符队列，随后将 ctor，dtor 和 name 参数初始化。

```
if (setup_cpu_cache(cachep)) {
  __kmem_cache_destroy(cachep);
  cachep = NULL;
  goto oops;
}

/* cache setup completed, link it into the list */
list_add(&cachep->next, &cache_chain);
```

这段函数使用 setup_cpu_cache 函数将 Cache 描述符的其他参数初始化，包括 array、nodelists、batchcount 和 limit 参数，之后将 Cache 描述符加入到全局链表 cache_chain 中。至此，Cache 描述符的创建完成。在 Linux 系统中，Cache 描述符一旦创建，一般不会被释放。

3. Cache 描述符的释放

在 Linux 系统中，通用 Cache 描述符一旦被创建后，将在整个系统的运行期间保持有效；绝大多数专用 Cache 描述符在创建后也不会被释放。但是还有个别专有 Cache 描述符，如在有些设备驱动程序中创建的 Cache 描述符，因为出现种种错误，而需要释放创建的 Cache 描述符。

如果在动态加载的设备驱动程序中使用了 Cache 描述符，那么在设备驱动程序卸载时，需要将所分配的 Cache 描述符释放。本书不建议一般用户在编写自己的设备驱动程序时使用这种方式，提高自定义数据结构的分配效率。因为绝大多数驱动程序没有复杂到必须用创建专用 Cache 描述符的方式进行优化，在多数情况下，程序员可以找到许多更加简练的方式优化设

备驱动程序。Slab 分配器使用 kmem _ cache _ destroy 函数释放 Cache 描述符。该函数的源代码如下：

```
void kmem_cache_destroy(struct kmem_cache * cachep)
{
  BUG_ON(! cachep || in_interrupt());
  /* Find the cache in the chain of caches. */
  mutex_lock(&cache_chain_mutex);

  list_del(&cachep->next);
  if (__cache_shrink(cachep)) {
    slab_error(cachep, "Can't free all objects");
    list_add(&cachep->next, &cache_chain);
    mutex_unlock(&cache_chain_mutex);
    return;
  }

  if (unlikely(cachep->flags & SLAB_DESTROY_BY_RCU))
    synchronize_rcu();

  __kmem_cache_destroy(cachep);
  mutex_unlock(&cache_chain_mutex);
}
```

kemem _ cache _ destrop 函数首先将 Cache 描述符从 cache _ chain 队列中摘除，之后使用 __ cache _ shrink 函数释放 Cache 描述符占用的空间。在__ cahce _ shrink 函数中，使用 kfree 函数将 array 参数中的空间释放，同时释放 Cache 描述符的 nodelists 参数占用的空间。最后该函数将存放在 cache _ cache 中的 Cache 描述符使用的空间释放。

7.3.3 Slab 的管理

Linux 系统将 Slab 连接成一个个链表，由 Cache 统一进行管理。由 7.3.1 节可知，每一个 Cache 描述符都有一个 nodelists 参数。在 nodelists 中，包含三个指向不同 Slab 链表的指针，分别是 slabs _ full、slabs _ partial 与 slabs _ free。这三个指针将在下文详细介绍。

Linux 系统还设置了 CFLGS _ OFF _ SLAB 标识，确定当前 Slab 描述符的存放方法。如果 CFLGS _ OFF _ SLAB 标识为 0，表示 Slab 描述符与 Slab 中的数据对象共享内存空间，否则不共享。其存放关系如图 7-11 所示。

如上图所示，采用这种方式时，Slab 描述符与其管理的 object 共享在 Buddy System 中的空间，可能是一个或者多个物理地址连续的页面。如果 CFLGS _ OFF _ SLAB 标识为 1，表示 Slab 描述符与数据对象不共享内存空间。其存放关系如图 7-12 所示。

1. Slab 的创建

由 7.3.2 节可知，Linux 系统使用 kmem _ cache _ create 函数创建 Cache 描述符时，将调用 kmem _ cache _ zalloc -> __ cache _ alloc -> ____ cache _ alloc -> cache _ alloc _ refill -> cache _

grow 函数为当前 Cache 创建新的 Slab。kmalloc 或者 kmalloc _ node 函数申请内存空间时，有时也会调用__ kmalloc ->__ do _ kmalloc ->__ cache _ alloc ->____ cache _ alloc ->cache _ alloc _ refill ->cache _ grow 函数为当前 Cache 创建新的 Slab。

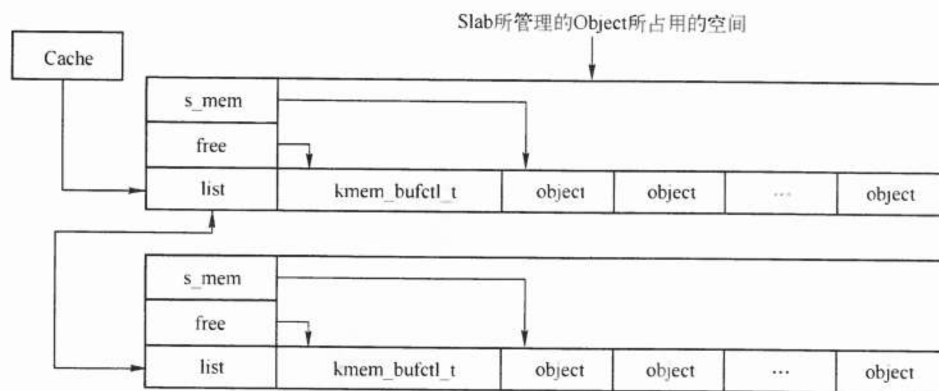


图 7-11　Slab 描述符与 Object 共享物理地址空间

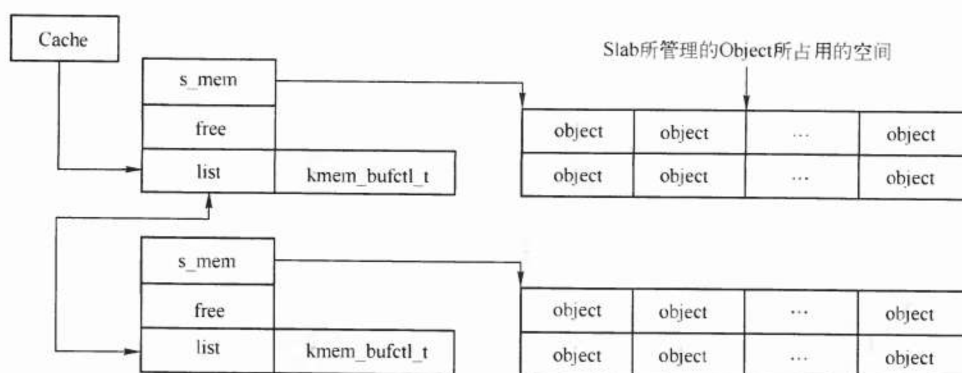


图 7-12　Slab 描述符不与 Object 共享物理地址空间

这些新生成的 Slab 将加入到 Cache 描述符的 slabs _ free 链表指针中。cache _ grow 函数的定义在 ./mm/slab.c 文件中。

```
static int cache _ grow(struct kmem _ cache * cachep, gfp _ t flags, int nodeid)
{
```

如果 cache _ grow 函数创建 Slab 成功则返回 1，如果失败则返回 0。cache _ grow 函数的主要作用是将初始化完毕的 Slab 加入到当前 Cache 的 slabs _ free 指针中。cache _ grow 函数一共有三个输入参数。

(1) cachep 参数表示为哪个 Cache 申请 Slab。

(2) flags 参数表示采用何种方式创建 Slab。

(3) nodeid 参数表示使用哪个内存节点保存 Slab 的物理页面。

```
...
    if (flags & __ GFP _ NO _ GROW)
```

```
    return 0;

    ctor_flags = SLAB_CTOR_CONSTRUCTOR;
    local_flags = (flags & GFP_LEVEL_MASK);
    if (!(local_flags & __GFP_WAIT))
      ctor_flags |= SLAB_CTOR_ATOMIC;
...
    offset *= cachep->colour_off;

    if (local_flags & __GFP_WAIT)
      local_irq_enable();

    kmem_flagcheck(cachep, flags);
```

这段程序对 cache_grow 函数的入口参数进行完整性检查,并根据 Cache 中的 colour_off 参数确定每一个 Slab 之间的间隔。

```
  /*
   * Get mem for the objs. Attempt to allocate a physical page from
   * 'nodeid'.
   */
  if (!objp)
      objp = kmem_getpages(cachep, flags, nodeid);
  if (!objp)
      goto failed;
```

这段程序调用 kmem_getpages 函数,为 Slab 中的数据对象在 Buddy System 中分配物理地址空间。Slab 分配器基于 Buddy System。在 Linux 系统中,Slab 分配器只有在申请新的 Slab 时才与 Buddy System 进行交互,在绝大多数情况下,kmalloc 函数并不与 Buddy System 进行交互。kmem_getpages 函数的输入参数与 cache_grow 函数的输入参数相同,返回值为所分配物理页面的虚拟地址,该函数的执行流程如下:

- 首先使用 alloc_pages_node 函数在 Buddy System 中,分配合适的物理页面。
- 为分配的物理页面赋予 PG_slab 标识,表示该物理页面已被 Slab 分配器使用。
- 最后使用 page_address 函数获得当前物理页面的虚拟地址。

```
  /* Get slab management. */
  slabp = alloc_slabmgmt(cachep, objp, offset,
      local_flags & ~GFP_THISNODE, nodeid);
  if (!slabp)
    goto opps1;
```

调用 alloc_slabmgmt 函数获得 Slab 描述符的空间。由上文所示,Slab 描述符与数据对象可以共享内存空间,也可以不共享内存空间。如果 Slab 描述符不与数据对象共享内存空间,alloc_slabmgmt 函数将调用 kmem_cache_alloc_node 函数,从 Buddy System 中为当前 Cache

的 Slab 描述符分配物理空间；否则 Slab 描述符将与数据对象共享物理空间。

```
    slabp->nodeid = nodeid;
    slab_map_pages(cachep, slabp, objp);

    cache_init_objs(cachep, slabp, ctor_flags);

    if (local_flags & __GFP_WAIT)
      local_irq_disable();
    check_irq_off();
    spin_lock(&l3->list_lock);

    /* Make slab active. */
    list_add_tail(&slabp->list, &(l3->slabs_free));
    STATS_INC_GROWN(cachep);
    l3->free_objects += cachep->num;
    spin_unlock(&l3->list_lock);
    return 1;
  opps1:
    kmem_freepages(cachep, objp);
  failed:
    if (local_flags & __GFP_WAIT)
      local_irq_disable();
    return 0;
  } /* End cache_grow */
```

这段函数的执行流程如下：

- 调用 slab_map_pages 函数，将存放 Slab 数据对象的物理页面的 lru.next 和 lru.prev 指向该数据对象所属的 Cache 描述符和 Slab 描述符。之后，Slab 分配器可以根据数据对象的虚拟地址，通过 virt_to_page 函数以及 page_get_cache 和 page_get_slab 函数，获得这个虚拟地址所对应的 Cache 描述符和 Slab 描述符。kfree 函数将使用 lru.next 和 lru.prev。
- 调用 cache_init_objs 函数将 Slab 中的数据对象进行初始化，并进行函数返回。如果 cache_grow 函数的返回值为 1，表示该函数成功地从 Buddy System 中获得了新的 Slab；如果该函数的返回值为 0，表示该函数执行失败。

在这里，需要简单介绍一下 cache_init_objs 函数。该函数的主要源代码如下所示：

```
static void cache_init_objs(struct kmem_cache *cachep,
          struct slab *slabp, unsigned long ctor_flags)
{
  int i;

  for (i = 0; i < cachep->num; i++) {
```

```
        void * objp = index_to_obj(cachep, slabp, i);
        if (cachep->ctor)
            cachep->ctor(objp, cachep, ctor_flags);
        slab_bufctl(slabp)[i] = i + 1;
    }
    slab_bufctl(slabp)[i - 1] = BUFCTL_END;
    slabp->free = 0;
}
```

cache_init_objs 函数的主要作用是初始化 kmem_bufctl_t 数组。

在 Slab 中，kmem_bufctl_t 数组用作空闲数据对象的索引。kmem_bufctl_t 数组的存放位置在 Slab 描述符之后，如图 7-11，7-12 所示。这段程序使用内联函数 slab_bufctl 访问 kmem_bufctl_t 数组，slab_bufctl 函数的定义为（kmem_bufctl_t *）(slabp + 1)。

这段函数根据在当前 Slab 中可以存放的数据对象个数确定 kmem_bufctl_t 数组的大小，并将 1，2，...，cachep->num，分别赋值到 kmem_bufctl_t[0～ cachep->num-1]中，然后将 BUFCTL_END 赋值到 kmem_bufctl_t[cachep->num]中。其中 BUFCTL_END 参数用来确定 kmem_bufctl_t 数组的结束状态。kmem_bufctl_t 数组中的每一位与 Slab 中的空闲数据对象相对应。

在这里，认真的读者读者应该注意到在 kmem_bufctl_t 数组中，并没有与 Slab 中第一个数据对象相对应的 Entry，即 kmem_bufctl_t 数组并没有存放 0 这个数值。因为 Linux 系统使用 Slab 描述符的 slabp->free 参数存放着第一个未使用的数据对象。在 Slab 初始化时，Slab 描述符的 free 参数被置为 0，Slab 的第一个空闲数据对象就是保存在这里。

Linux 系统使用 Slab 分配器进行内存申请时，Slab 中的数据对象将会不断地被分配，slabp->free 参数将被依次赋值为 kmem_bufctl_t[0～ cachep->num-1]。

2. Slab 的释放

在 Linux 系统中，Slab 轻易不会被释放。只有在存放 Slab 的 Cache 空间被释放或者当操作系统由于物理内存空间紧张而对 Cache 空间进行整理时，才有可能释放 Slab 占用的物理空间。这种做法保证了 Slab 在 Linux 系统中长期有效，从而保证了 Linux 系统进行内存申请的效率。在 Linux 系统中使用 slab_destroy 函数释放 Slab 所占用的物理空间。这个函数的源代码在 ./mm/slab.c 文件中。该函数源代码详解如下：

```
static void slab_destroy(struct kmem_cache * cachep, struct slab * slabp)
{
    void * addr = slabp->s_mem - slabp->colouroff;

    slab_destroy_objs(cachep, slabp);
    if (unlikely(cachep->flags & SLAB_DESTROY_BY_RCU)) {
        struct slab_rcu * slab_rcu;

        slab_rcu = (struct slab_rcu * )slabp;
```

```
        slab_rcu->cachep = cachep;
        slab_rcu->addr = addr;
        call_rcu(&slab_rcu->head, kmem_rcu_free);
    } else {
        kmem_freepages(cachep, addr);
        if (OFF_SLAB(cachep))
            kmem_cache_free(cachep->slabp_cache, slabp);
    }
}
```

- 如果被释放的 Slab 具有析构函数，则 slab_destroy_objs 函数调用此析构函数，将 Slab 中存放的所有数据对象进行析构操作。
- 使用 kmem_freepages 函数释放 Slab 所占用的内存空间。
- 如果当前 Slab 描述符不与数据对象共享内存空间，还需要使用 kmem_cache_free 函数释放 Slab 描述符所占用的内存空间。

7.3.4 数据对象的管理

数据对象与 Slab 的关系如图 7-11 和 7-12 所示。在 Linux 系统中，数据对象分为两类：存放在专用 Cache 中的数据对象被称为专用数据对象，而存放在通用 Cache 中的数据对象被称为通用数据对象。本节将详细介绍通用数据对象的创建与释放。

1. 数据对象的创建

Linux 系统使用 kmem_cache_alloc 函数创建专用数据对象；使用 kmalloc 函数创建通用数据对象。通用数据对象是 Linux 系统动态申请的内存空间，专用数据对象存放 Linux 系统中常用的数据结构，如上文提到的 task_struct 结构。在 Linux 系统中，通用数据对象的创建是专用数据对象创建的一个特例，通用数据对象的创立也需要调用 kmem_cache_alloc 函数。kmalloc 函数用作内存申请。该函数的源代码在 ./include/linux/slab_def.h 文件中，其详解如下：

```
static inline void * kmalloc(size_t size, gfp_t flags)
{
```

该函数共有两个输入参数，分别为 size 和 flags。size 参数描述 kmalloc 函数所申请的空间大小，而 flags 参数与 7.2.3 节 gfp_mask 参数的定义相同。该函数成功返回后，将得到所申请空间的虚拟地址，否则返回值为 NULL。

```
    if (__builtin_constant_p(size)) {
        int i = 0;
#define CACHE(x) \
        if (size <= x) \
            goto found; \
        else \
            i++;
```

```
#include "kmalloc_sizes.h"
#undef CACHE
    {
      extern void __you_cannot_kmalloc_that_much(void);
      __you_cannot_kmalloc_that_much();
    }
found:
    return kmem_cache_alloc((flags & GFP_DMA) ?
      malloc_sizes[i].cs_dmacachep :
      malloc_sizes[i].cs_cachep, flags);
  }
```

这段程序使用__builtin_constant_p函数，判断kmalloc函数的输入参数size是否被Gcc编译器判定为一个常数。如果size参数为一个常数，则__builtin_constant_p函数为True，否则为False。程序员使用kmalloc(128, flags)这样的语句(此时size参数为128，是一个常数，而不是一个变量)，进入kmalloc函数时，__builtin_constant_p(size)的返回值为真。

__builtin_constant_p是Gcc编译器使用的函数，该函数帮助程序判断一个表达式是否在编译阶段就被确定为常数。该函数对一些性能要求较高的程序有一定帮助。

之后这段程序重新定义宏Cache，然后在全局数组malloc_sizes中，查找与size参数对应的Entry。这段程序写得比较精炼，希望读者认真理解这段代码。

如果在全局数组malloc_sizes中，没有查找到与size参数对应的Entry，则调用__you_cannot_kmalloc_that_much函数；如果在kmalloc_sizes.h文件中找到了合适的Entry，则调用kmem_cache_alloc函数申请内存空间。

```
    return __kmalloc(size, flags);
  }
```

如果kmalloc函数的输入参数size被Gcc编译器判定是一个变量，则调用__kmalloc函数申请内存空间。kmem_cache_alloc函数和__kmalloc函数的执行流程如图7-13所示。

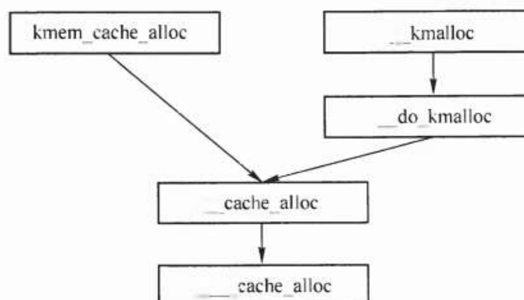


图7-13 与数据对象创建有关的函数

由上图所示，kmem_cache_alloc和__kmalloc函数最终调用____cache_alloc函数完成通用或者专用数据对象的创建。____cache_alloc函数如下：

```
static inline void * ____cache_alloc(struct kmem_cache * cachep, gfp_t flags)
{
    void * objp;
    struct array_cache * ac;

    check_irq_off();

    if (should_failslab(cachep, flags))
        return NULL;

    ac = cpu_cache_get(cachep);
    if (likely(ac->avail)) {
        STATS_INC_ALLOCHIT(cachep);
        ac->touched = 1;
        objp = ac->entry[--ac->avail];
    } else {
        STATS_INC_ALLOCMISS(cachep);
        objp = cache_alloc_refill(cachep, flags);
    }
    return objp;
}
```

这段源代码的执行流程如下：

(1) 调用cpu_cache_get获得当前Cache的array参数,之后调用cpu_cache_get函数试图从Cache的array参数中获得数据对象。

(2) 如果在当前Cache描述符的array中有空闲的数据对象缓存,则获得此数据对象。否则调用cache_alloc_refill函数重新装填当前Cache的array缓存,之后从array中获得相应的数据对象。

cache_alloc_refill函数是____cache_alloc函数的主体。该函数的返回值是数据对象的虚拟地址。该函数的源代码在./mm/slab.c文件中,其执行流程如下：

(1) 进行基本的参数检查,并获得当前Cache的array参数。

(2) 获得当前Cache的nodelists参数,判断是否在nodelists的共享array中存在数据空间。如果有,则将在共享array中的数据空间搬移到Cache的array参数中,然后获得此数据空间,并转移到第6步。

(3) 根据batchcount参数的大小决定一共需要从当前Cache的nodelists中的Slab链表获得多少个数据对象。首先此函数将检查nodelists的slabs_partial链表,如果此链表不为空则从此链表中获得数据对象。否则继续检查nodelists的slabs_free链表是否为空,如果此链表不为空则从此链表中获得数据对象,否则转到第5步。

(4) 使用slab_get_obj函数,从nodelists的slabs_partial链表或者slabs_free链表中获得数据对象,并装填到当前Cache的array中。slab_get_obj函数将调用slab_bufctl函数维护Slab中的kmem_bufctl_t数组。转到第6步。

(5) 使用 cache _ grow 函数创建新的 Slab,并将其加到 nodelists 的 slabs _ free 链表中。

(6) 从 array 中获得数据对象然后返回。

____ cache _ alloc 函数的实现过程并不十分完美,有些程序书写得很冗长,这段程序的结构也存在着一些问题。但其仍不失为一段好程序。该函数的源代码较长,本书为节约篇幅不再列出,希望读者务必与其源代码对应,仔细理解该函数的执行流程。

Linux 系统主要使用 kmem _ cache _ alloc 函数创建专用数据对象。但也经常使用一些"封装过"的函数。如创建专用数据对象 task _ struct 时,Linux 系统使用 alloc _ task _ struct 函数,该函数将调用 kmem _ cache _ alloc 函数从专用 Cache 描述符 task _ struct _ cachep 中分配一个 task _ struct 结构。

2. 数据对象的释放

Linux 系统使用 kmem _ cache _ free 和 kfree 函数,释放数据对象。这两个函数最终调用 __ cache _ free 函数释放数据对象。__ cache _ free 函数与 kmem _ cache _ free 函数的输入参数相同,分别为 cachep 和 objp。其中 cachep 参数指向 Cache 描述符,而 objp 参数指向将要被释放的数据对象。

kfree 函数的输入参数只有一个,即所释放数据对象的虚拟地址的指针 objp。在 kfree 函数中,首先使用 virt _ to _ cache 函数根据虚拟地址获得 objp 的物理页面指针,然后通过该物理页面指针获得数据对象所在的 Cache 描述符,最后调用__ cache _ free 函数释放数据对象。__ cache _ free 函数的执行流程如下:

- 调用 cpu _ cache _ get 函数获得当前 Cache 的 array 参数。
- 如果当前 Cache 的 array 中还有空间,则将数据对象直接释放到 array 中,完成数据对象的释放。
- 如果当前 Cache 的 array 中没有空间,则调用 cache _ flusharray 函数,重新整理 array 中的数据对象,将 array 中的数据对象加入到 Cache->nodelists->slabs _ free 链表中。随后完成数据对象的释放。

至此我们讲述了数据对象的创建与释放,从上述过程中可以看到,Slab 分配器中使用了许多缓存以提高数据对象管理的效率。在绝大多数情况下,Linux 系统创建与释放数据对象的效率都非常高。

7.4 VM 空间

从上文中,可以发现如果用户使用 Buddy System,进行内存申请,可以获得的最大内存是 2^{max_order}个页面,即 16 MB;如果使用 Slab 分配器,可以获得的最大内存是 2^{17}字节,即 128 KB。

由此可见,需要一段相对较大的、物理地址连续的内存空间如 2 MB 时,可以使用 Buddy System 或者 Boot Memory,但是在 Buddy System 和 Boot Memory 中,物理地址连续的空间十分珍贵,因此 Linux 系统提供了一种特殊的内存分配方式,即不连续内存分配。

在 Linux 系统中,使用 vmalloc 和 vfree 函数进行这种内存的申请与释放。Linux 系统将这段空间放在宏 VMALLOC _ START 与 VMALLOC _ END 之间。vmalloc 函数将使用这段在 VMALLOC _ START 和 VMALLOC _ END 之间的空间,进行内存分配,下文将这段空间称为 VM 空间。Linux PowerPC 在 ./include/asm-ppc/pgtable.h 文件中定义这两个宏:

```
#define VMALLOC_OFFSET    (0x1000000)    /* 16M */
#define VMALLOC_START                          \
    ((((long)high_memory + VMALLOC_OFFSET) & ~(VMALLOC_OFFSET-1)))
#define VMALLOC_END       ioremap_bot
```

由以上定义，可以发现宏 VMALLOC_START 与 high_memory 参数有关，其值 16 MB 对界。在 Linux 系统中，high_memory 是一个全局变量，保存当前系统的高端内存的起始地址。如果 Linux 系统使能了 HIMEM 空间，而且其物理内存大于或者等于 768 MB 时，宏 VMALLOC_START 的值为 0xF100-0000；如果一个处理器没有使用 HIMEM 空间，而且其物理内存小于 768 MB 时，如当前处理器系统只有 256 MB 时，宏 VMALLOC_START 的值为 0xD100-0000(256 MB+16 MB+0xC000-0000)。

宏 VMALLOC_END 的值与 ioremap_bot 参数有关，ioremap_bot 参数保存 ioremap 函数使用的虚拟地址空间尾部，该变量在 ./arch/powerpc/mm/init_32.c 文件中初始化。

```
#ifdef CONFIG_HIGHMEM
  ioremap_base = PKMAP_BASE;
#else
  ioremap_base = 0xfe000000UL; /* for now, could be 0xfffff000 */
#endif /* CONFIG_HIGHMEM */

  ioremap_bot = ioremap_base;
```

如果 Linux 系统使用 HIMEM 空间，则 ioremap_base 变量为 PKMAP_BASE，否则该变量为 0xfe00-0000。在 Linux PowerPC 中，PKMAP_BASE 在 ./include/asm-ppc/highmem.h 文件中定义，其值为 0xfe00-0000。目前在 ./include/asm-powerpc 目录中还没有 highmem.h 文件。

程序员在使用 VM 空间时，需要十分谨慎，一般只推荐在 Swap 对换区和设备驱动程序中，对性能要求不高但是所占空间相对较大的内存中使用。因为使用 vmalloc 分配的空间的分配、释放与使用 Buddy System 或者 Slab 分配器管理的内存相比，效率较低。

VM 空间由全局指针 vmlist 统一管理，该指针指向一个 vm_struct 结构的数据，该结构的定义在 ./include/linux/vmalloc.h 文件中，其源代码如下：

```
struct vm_struct {
  /* keep next,addr,size together to speedup lookups */
  struct vm_struct *next;
  void          *addr;
  unsigned long    size;
  unsigned long    flags;
  struct page     **pages;
  unsigned int    nr_pages;
  unsigned long    phys_addr;
};
```

在 VM 空间中，有一段非常重要的空间，外部设备驱动程序的 ioremap 函数使用 VM 空间，实现外部设备物理地址到虚拟地址的转换。ioremap 函数调用__ioremap 函数完成物理地址到虚拟地址的转换。__ioremap 函数在./arch/powerpc/mm/pgtable_32.c 文件中定义。其源代码如下：

```
void __iomem *
__ioremap(phys_addr_t addr, unsigned long size, unsigned long flags)
{
...
  p = addr & PAGE_MASK;
  size = PAGE_ALIGN(addr + size) - p;

  if (p < 16 * 1024 * 1024)
    p += _ISA_MEM_BASE;

  if (mem_init_done && (p < virt_to_phys(high_memory))) {
    printk("__ioremap(): phys addr "PHYS_FMT" is RAM lr %\n", p,
        __builtin_return_address(0));
    return NULL;
  }
  if (size == 0)
    return NULL;
```

__ioremap 函数共有三个输入参数，分别为 addr、size 和 flags 参数。addr 参数用来保存需要进行映射的物理地址；size 参数用来保存需要映射的空间大小；flags 参数用来描述映射空间的页面属性。

__ioremap 函数程序首先对参数进行检查，检查当前物理地址是否在 ISA 总线的地址空间中，在 Linux PowerPC 中并没有 ISA 总线空间。

__ioremap 函数在 Buddy System 初始化完毕之后(即 mem_init_done 参数为 1 后)，其物理地址 p 对应的虚拟地址不能大于 high_memory，即不能占用 HIMEM 空间。

```
if ((v = p_mapped_by_bats(p)) /* && p_mapped_by_bats(p+size-1) */)
  goto out;

if ((v = p_mapped_by_tlbcam(p)))
  goto out;
```

这段程序首先使用 p_mapped_by_bats 函数，试图在 DBAT 中进行物理地址的映射。但是 E500 内核不支持 DBAT，因此对于 E500 内核，使用该函数一定无法获得虚拟地址。基于 E500 内核的 Linux PowerPC，p_mapped_by_bats 函数的返回值为 0。

然后这段程序继续调用 p_mapped_by_tlbcam 函数，试图在 TLB1 中，进行物理地址的映射，如果系统程序员没有使用 7.1.1 节中的 io_block_mapping 函数事先进行虚实地址映射，则该函数不会获得虚拟地址。

在 Linux 系统中，io _ block _ mapping 函数所能映射的空间，应该在 VM 空间之内。但是有一些系统程序员并没有这样做，这也是 Linux 系统维护者取消 io _ block _ mapping 函数的重要原因。Linux 系统可以支持各式各样的应用，并将这些应用的共性集中放到 Linux 内核中。也是因为这个原因，一个标准的 Linux 系统，也许不能在一个特定的应用中，发挥最大的效率。

```
if (mem_init_done) {
    struct vm_struct *area;
    area = get_vm_area(size, VM_IOREMAP);
    if (area == 0)
        return NULL;
    v = (unsigned long) area->addr;
} else {
    v = (ioremap_bot -= size);
}
```

这段程序是__ ioremap 函数最重要的代码。在 Linux PowerPC 的标准发布版本中，并没有使用 io _ block _ mapping 函数。因此在 Linux PowerPC 中，物理地址到虚拟地址进行映射时，必须使用这段代码。这段代码中首先对 mem _ init _ done 参数进行判断。

- 如果 mem _ init _ done 参数为 1，即 mem _ init 函数执行完毕，Buddy System 已经初始化完毕。此时__ ioremap 函数调用 get _ vm _ area 函数在 VM 空间获得一段虚拟地址。Linux 系统使用 vm _ struct 结构管理 VM 空间，而 vm _ struct 结构由 kmalloc 函数进行分配。使用 get _ vm _ area 函数时，mem _ init _ done 变量必须为 1。
- 如果 mem _ init _ done 为 0，即 mem _ init 函数没有执行完毕，此时只能从 VM 空间的尾部预留虚拟地址空间，虚拟空间分配完毕后将调整 VM 空间的大小。由上文得知宏 VMALLOC _ END 等效于 ioremap _ bot。在 Linux 系统中，setup _ arch 函数和 init _ IRQ 函数中也会使用 ioremap 函数，此时 Buddy System 并没有初始化完毕，因此必须使用这种方式，建立物理地址到虚拟地址的映射。

```
if ((flags & _PAGE_PRESENT) == 0)
    flags |= _PAGE_KERNEL;
if (flags & _PAGE_NO_CACHE)
    flags |= _PAGE_GUARDED;

/*
 * Should check if it is a candidate for a BAT mapping
 */

err = 0;
for (i = 0; i < size && err == 0; i += PAGE_SIZE)
    err = map_page(v+i, p+i, flags);
if (err) {
    if (mem_init_done)
```

```
            vunmap((void * )v);
        return NULL;
    }

out:
    return (void __ iomem * ) (v + ((unsigned long)addr & ~PAGE_MASK));
} /* End __ ioremap */
```

这段程序首先对 flags 参数进行互斥检查,之后调用 map _ page 函数,建立 pte 表,将获得的虚拟地址与物理地址联系在一起。如果用户使用 TLB1 进行虚实地址映射时,不需要使用 pte 表。最后这段程序将获得虚拟地址返回。

7.5 HIMEM

HIMEM 占用内核 PKMAP _ BASE～0xFFFFF000 之间的虚拟空间。在 Linux PowerPC 中 PKMAP _ BASE 为 0xFE000000,因此这段空间的大小为 32764KB。所有的高端内存,即大于 768MB 的内存,需要映射到这段空间后,才能被 Linux 内核访问。

Linux 系统为 HIMEM 提供了两个函数 kmap 和 kunmap,kmap 函数用来将高端内存映射到低端,而 kunmap 函数将解除这种空间映射。

对于多数基于 PowerPC 处理器的应用,其物理内存一般小于 768MB,因此不需要使用 HIMEM。而拥有较大内存的服务器一般会使用 64 位 PowerPC 处理器,因此 HIMEM 实际上并没有太大的用武之地。

有许多 Linux 系统程序员,采用改变 CONFIG _ KERNEL _ START 参数的方法对 Linux 系统中超过 768 MB 的物理内存进行管理。本书并不建议采用这种方式。因为当用户改变了 CONFIG _ KERNEL _ START 参数后,会对整个 Linux 内核的内存管理造成一定的影响,虽然这种影响并不大。

Linux 系统内存管理的系统设计者考虑到,有人会改写 CONFIG _ KERNEL _ START 参数,从而采取了一些相应的措施,但是这些措施并没有被充分地测试。

Linux 系统对 HIMEM 的管理的实现与 Slab 分配器的实现相比较为简单,也比较容易分析,希望读者能够自行分析这些代码。许多系统程序员对 Linux 系统的 HIMEM 较为排斥。在多数情况下,他们不喜欢一个系统中,因为先天不足而后加入的特性。

7.6 进程地址空间

第 5 章提到,在 Linux 系统中,进程分为核心进程与用户进程。其中核心进程与 Linux 内核共享正文段与数据段,因此核心进程的地址空间在 Linux 内核的地址空间中,而用户进程占用 Linux 系统 0～3 GB 这段虚存空间。提醒读者注意,Linux 系统的每一个用户进程都有各自的 3 GB 虚存空间,在其运行时都将使用 0～3 GB 这段虚存空间。Linux 进程地址空间的设计较为复杂,本书将简单介绍这部分内容。

7.6.1 进程的内存描述符

Linux 系统使用 mm _ struct 结构，对用户进程的地址空间进行描述。由第 3 章所述，每一个进程都有一个进程描述符 task _ structs，其中 task _ structs->mm 指向各自进程的内存描述符 mm _ struct。在进程进行切换时，也需要对 mm _ struct 参数进行切换。数据结构 mm _ struct 的定义在 ./include/linux/sched.h 文件中，其主要数据成员如下所示：

```
struct mm_struct {
  struct vm_area_struct * mmap; /* list of VMAs */
  struct rb_root mm_rb;
  pgd_t * pgd;
  struct list_head mmlist;
  unsigned long start_code, end_code, start_data, end_data;
  unsigned long start_brk, brk, start_stack;
  unsigned long arg_start, arg_end, env_start, env_end;
}
```

(1) mmap 参数。用户进程一共使用 3GB 大小的空间。而实际上，大多数进程不可能使用这么大的内存空间，因此这些空间使用数据结构 vm _ area _ struct 分段进行管理。mmap 参数将这些空间连接在一起。

(2) mm _ rb 参数。普通进程只有几个使用 mmap 参数连接在一起的内存空间段，有关这些内存空间段的详细信息可以参阅/proc/ID/map 文件。对于绝大多数进程，采用什么方式组织这些内存空间段并没有太大关系。但是在 Linux 系统中，存在某些数据库应用，其中包含了许多内存空间段，因此需要使用某种快速算法用来对当前进程的所有内存空间段进行管理。

在 Linux 系统中，find _ vma 函数将经常被使用，这个函数的主要作用是查找当前进程的虚拟地址所在的内存空间段。这个函数使用 mm _ rb 参数，对当前进程的所有内存空间段进行扫描，其扫描速度决定了进程地址空间的管理效率。

为此 Linux 使用红黑树结构将这些内存空间段组织管理起来，mm _ rb 参数是 Linux 进程地址空间管理，实现红黑树算法的数据结构。本书不对红黑树算法进行讨论。对此有兴趣的读者可以参考 Chris Okasaki 所著的 Purely Functional Data Structures。

(3) pgd 参数。每一个进程都有自己的 pgd 指针，指向各自进程的 PGD 的基地址。不同的进程使用不同的 pgd 参数，进行虚实地址的转换。Linux 系统使用 pgd 参数将每一个用户进程的地址空间分开，虽然这些用户进程都使用 0～3GB 的虚拟地址空间。

(4) mmlist 参数。这个参数将所有的 mm _ struct 数据结构链接在一起。

(5) start _ code, end _ code 参数。这组参数描述当前进程正文段的起始与结束地址。

(6) start _ data, end _ data 参数。这组参数描述当前进程数据段的起始与结束地址。

(7) start _ brk, brk 参数。这组参数描述当前进程数据堆段的起始与结束地址。

(8) start _ stack 参数。这组参数描述当前进程栈段的起始地址。

(9) arg _ start, arg _ end 参数。这组参数描述当前进程存放的命令行参数的起始与结束地址。

(10) env _ start, env _ end 参数。这组参数描述当前进程存放的 C 环境变量的起始与结

束地址。

1. 内存描述符的创建与初始化

在 Linux PowerPC 中,第一个内存描述符 mm_strcut 是全局变量 init_mm 即进程 0 的 active_mm 参数,即 init_task->active_mm 被初始化为 init_mm。

init_mm 变量的定义在 ./arch/powerpc/kernel/init_task.c 文件中。在这个文件中,使用宏 INIT_MM 对此全局变量进行初始化。

```
struct mm_struct init_mm = INIT_MM(init_mm);
#define INIT_MM(name)                                                        \
{
    .mm_rb        = RB_ROOT,                                                 \
    .pgd          = swapper_pg_dir,                                          \
    .mm_users     = ATOMIC_INIT(2),                                          \
    .mm_count     = ATOMIC_INIT(1),                                          \
    .mmap_sem     = __RWSEM_INITIALIZER(name.mmap_sem),                      \
    .page_table_lock = __SPIN_LOCK_UNLOCKED(name.page_table_lock),           \
    .mmlist = LIST_HEAD_INIT(name.mmlist),                                   \
    .cpu_vm_mask = CPU_MASK_ALL,                                             \
}
```

除 init_mm 内存描述符之外,其他内存描述符都在新进程创建时由 do_fork 函数创建。内存描述符的创建流程如图 7-14 所示。这里提醒读者注意,Linux 系统的核心进程和整个 Linux 内核共享内存空间,因此没有内存描述符,这类进程描述符的 task_struct->mm 参数为 NULL。

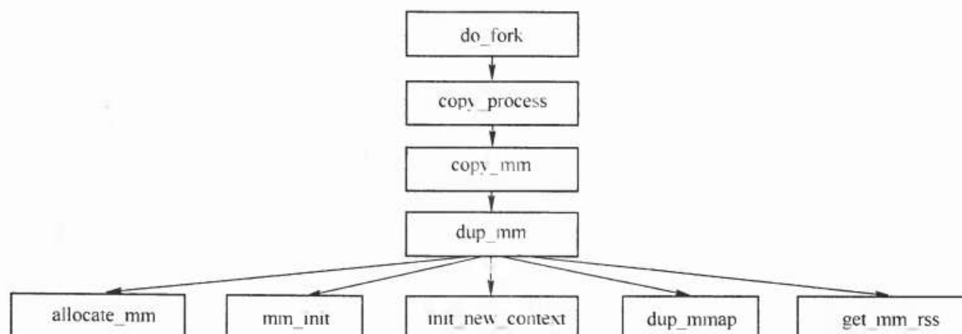


图 7-14 内存描述符的创建

如上图所示,Linux 系统使用 dup_mm 函数创建进程的内存描述符。dup_mm 函数的源代码在 ./kernel/fork.c 文件中,其执行流程如下:

- 首先调用 allocate_mm→kmem_cache_alloc 函数,从 mm_cachep(专用 Cache)中,获得 mm_struct 所需要的空间。为了提高 mm_struct 结构的访问速度,Linux 系统使用专用 Cache mm_cachep,存放 mm_struct 结构,而不使用 kmalloc 函数申请 mm_struct 使用的内存空间。
- 调用 mm_init 函数将 mm_struct 结构的基本数据成员进行初始化。mm_init 函数调

用 mm _ alloc _ pgd 函数，为当前内存描述符分配 PGD 表。在 Linux 系统中，每一个用户进程都有独立的 PGD 表，每一个用户进程都有独立的进程虚拟地址空间。

- 调用 init _ new _ context，dup _ mmap 和 get _ mm _ rss 函数将 mm _ struct 结构的其他参数初始化。

2. 内存描述符的释放

Linux 系统在进程结束时，释放进程内存描述符，进程内存描述符的释放基本上是内存描述符创建的逆过程，其调用顺序如图 7-15 所示。

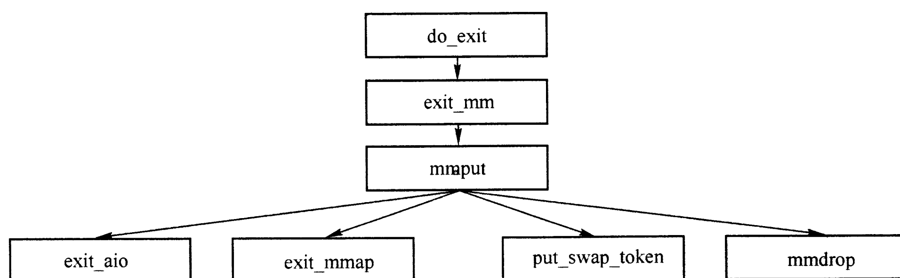


图 7-15　内存描述符的释放

Linux 系统调用 mmdrop -> mm _ free _ pgd 函数，释放进程的 PGD 表，最后调用 free _ mm 函数将内存描述符彻底释放。本书不再讲述这段程序的细节。

7.6.2　与进程地址空间有关的系统调用

在 Linux 系统中，用户可以使用一系列系统调用，与进程地址空间进行交互，不过多数系统调用对于 Linux 系统的使用者并没有太大帮助，可能有些程序员从来没有使用过这些系统调用。

1. 系统调用 brk

brk 是使用最为频繁的系统调用，用户进程可以通过该系统调用，向 Linux 内核申请内存空间。该系统调用可以调整用户进程堆空间的大小，使堆扩展或者缩小。Linux 系统使用 sys _ brk 函数实现此系统调用。当用户的堆空间过小时，可以使用该系统调用扩大堆空间；用户可以使用该系统调用释放内存空间。然而大多数应用程序都没有直接使用 brk 系统调用申请和释放内存空间，而是使用 malloc 和 free 函数申请和释放内存空间。

应用程序为 malloc 和 free 函数维护了一个 Buffer。当应用程序执行 malloc 函数时，如果该 Buffer 中有空闲的内存空间，则从 Buffer 中获得内存空间；如果 Buffer 中没有空闲的内存空间或者剩余的内存空间小于一个阈值，malloc 函数将调用 brk 系统调用向 Linux 内核申请内存空间，同时装填 Buffer。当应用程序执行 free 函数时，将释放的内存空间直接放入 Buffer 中，当 Buffer 中的内存空间超过一个阈值后，free 函数将调用 brk 系统调用将内存空间归还给 Linux 内核。

在 Linux 2.4 的内核中，sys _ brk 函数有个漏洞，该函数在调整用户进程的堆空间时，没有对参数 len 进行检查，也没有对 brk 函数的 addr + len 是否超过 TASK _ SIZE 做检查。这样黑客就可以提交任意大小的参数 len，改变用户进程的大小，甚至可以超过 TASK _ SIZE 参数的限制，使 Linux 系统认为内核使用的内存空间也可以被用户访问。这样普通用户就可以访问

内核使用的内存区域，之后通过堆溢出操作，获得管理员权限。

2. 系统调用 execve

execve 系统调用改变当前进程的地址空间，将之前继承的父进程的地址空间更换为系统调用 execve 指定的进程地址空间。Linux 系统使用 do_execve 函数实现此系统调用。在 Linux 2.4 中，这个函数的实现也有些漏洞，execve 系统调用允许进程在执行目标程序之前，使用 clone 系统调用复制一个进程，读取被执行的程序文件。攻击者可以利用此漏洞，读取正常情况下无法读取的文件，目前还没有黑客利用这个漏洞来攻击 Linux 系统的实例。

3. 系统调用 mmap，mmap2

mmap，mmap2 系统调用可以把文件映射到内存中。在 Linux 系统中，使用 sys_mmap 函数和 sys_mmap2 函数实现此系统调用。

最快的磁盘访问往往慢于最慢的物理内存访问，所以使用 mmap 系统调用，可以加速文件的 I/O 访问速度。此外 mmap 系统调用可以使进程通过映射相同的文件，实现共享内存。使用这类系统调用后，各进程可以像访问普通内存一样对文件进行访问。值得注意的是，具有直接血缘关系的进程间，也可以使用匿名文件进行内存映射，而不需要具体的文件句柄。

4. 系统调用 mremap

应用程序使用 mremap 系统调用改变内存空间段的边界地址。在 Linux 系统中，使用 do_mremap 函数实现此系统调用。

do_munmap 函数首先要清除在新位置中任何已经存在的内存映射，也就是删除旧的内存空间段，然后再建立新的内存空间段。这段代码没有对 do_munmap()函数的返回值进行检查，因此如果可用 vm_area_struct 描述符的最大数已经超出，那么函数调用可能会失败。黑客可以巧妙地利用这一漏洞，使用系统调用 mremap 对 Linux 系统进行攻击。攻击的详细方法见 http://isec.pl/vulnerabilities/isec-0014-mremap-unmap.txt。

5. 系统调用 munmap

munmap 系统调用是系统调用 mmap 的逆过程。在 Linux 内核中，使用 munmap 函数实现这一系统调用。Linux 系统还有 fork，exit，shmat，shmdt 等一系列系统调用，这些系统调用也可以与 Linux 系统的进程地址空间进行交互，本书对此将不一一叙述。

7.6.3 用户进程地址空间与 Linux 内核之间的数据交换

用户编写设备驱动程序时，需要交换用户进程地址空间与 Linux 内核间的数据。在 Linux 系统中，用户进程地址空间存放的数据并不可靠，因为这些数据并不能保证一定在物理内存中有效。

因为 Linux 系统为了保证物理内存的利用率，不可能将用户进程地址空间中的所有数据都存放到物理内存中。为此 Linux 系统提供一系列函数，进行用户进程地址空间与 Linux 内核之间的数据交换。这些函数在 ./include/asm-powerpc/uaccess.h 文件中。

某些设备驱动程序需要保证其访问的数据必须在物理内存中，如进行 DMA 传送时，DMA 控制器将直接访问物理内存，DMA 控制器并不知道其访问的数据是否在物理空间内，因为 DMA 控制器无法对此进行检查。但是 DMA 控制器要求其访问的数据必须在有效的物理地址空间中。

在 Linux 系统中，用户进程地址空间与 Linux 内核之间的进行数据交换时，经常使用 copy

_ from _ user 和 copy _ to _ user 函数。函数 copy _ from _ user 将用户进程地址空间的数据复制到内核空间;copy _ to _ user 函数将内核空间的数据复制到用户进程地址空间。这两个函数调用顺序如图 7-16 所示。

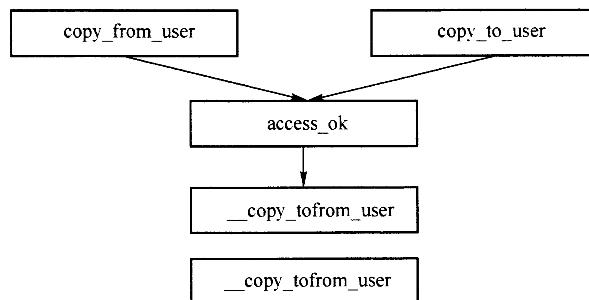


图 7-16 进程地址空间与 Linux 内核空间数据交换使用的函数

这两个函数的实现较为简单,首先调用 access _ ok 函数,判断相应数据空间的读写属性是否合法,之后调用__ copy _ tofrom _ user 函数,进行内存搬移。在 Linux PowerPC 中,__ copy _ tofrom _ user 函数在 ./arch/powerpc/lib/copy _ 32.S 文件中。

__ copy _ tofrom _ user 函数使用指令 lwzu,stwu 进行数据复制,这一函数的实现考虑了 L1 Cache 的利用率,因此这段程序的实现较为复杂。可能这段程序是使用 C 语言进行编写,然后再反汇编出来的,结构看起来并不好。

值得提醒读者注意,在__ copy _ tofrom _ user 函数进行数据复制时,有时用户进程地址空间的数据并不在物理内存中,因此在执行这段程序时有可能不断产生缺页异常,将在用户进程地址空间的数据放入物理内存,然后再执行数据复制指令。

由上文得知,用户进程地址空间使用 PowerPC 处理器的 TLB0 实现,而内核空间使用 TLB1 实现,因此 Linux 系统访问内核空间的数据时,不会出现 TLB Miss 中断,而 Linux 系统访问用户空间的数据时会出现 TLB Miss 中断。

设备驱动程序的设计者,有时为了提高效率,不希望进行用户进程地址空间与 Linux 内核之间的数据交换,此时有两种做法,可供读者参考。一种方法是在用户进程中使用 mlock 函数或者 mlockall 函数锁定用户进程地址空间,将用户进程的某段或者全部地址空间驻留在内存中。当用户进程地址空间不需要锁定时可以使用 munlock 函数和 munlockall 函数解除锁定。而另一种方法是在设备驱动程序中使用 sys _ mlock 函数将其使用的用户空间锁定,然后再进行访问,在访问结束后再使用 sys _ munlock 函数将这段空间释放。

这里有一个问题,需要提醒读者注意。在用户进程中的数据空间只保证一个页面内的数据物理地址连续。因此对于任何一段空间,即便只有 32 个字节,也可能是存放在两个不同的物理页面中。如果在设备驱动程序中使用 DMA 进行数据传送时,需要将这些数据分成两次进行传递。

在 Linux 系统中,我并没有从理论上探讨,在用户进程地址空间的数据区域是否都是 32 位对界的,但是我仍然清晰地记得基于 Alpha21264 的 OSF 操作系统中的进程地址空间内的有些数据区域居然不是 32 位对界。

当时我们的设备驱动程序为了提高效率,使用了页面锁定的技术,但是我们认为所有的数

据区域都是 32 位对界的。我们认为 Alpha21264 是一个 64 位处理器,OSF 也是针对此处理器的 64 位操作系统,所以所有的数据区域至少是 32 位对界。然而由此造成的程序错误,耽误了我们整个项目组将近一个月的时间。

因此我在 Linux 系统中使用页面锁定操作时,一定会检查数据区域是否 32 位对界,虽然至今为止,我还没有碰到数据区域不是 32 位对界的情况。

7.7 DSI/ISI 异常在 Linux PowerPC 中的处理

Linux 系统并不信任用户进程,并随时准备处理来自应用程序的内存访问错误。内存访问错误发生在以下两种情况中。

(1) 因为用户程序的执行错误而导致的内存访问错误。

(2) Linux 系统为完成某种功能而故意制造的内存访问错误。

Linux PowerPC 将这些内存访问错误分为进程数据空间访问错和进程程序空间访问错。E500 内核也提供了两种异常,处理这两种内存访问错误,分别为数据访问异常 DSI(Data Storage Interrupt)和指令访问异常 ISI(Instruction Storage Interrupt)。本书主要介绍 DSI 异常处理。

PowerPC E500 内核产生 DSI 异常的主要原因如下:

- 读取一些在 MMU 中不能进行读取的地址空间,向 MMU 中不能写入的地址空间进行写操作。Linux 系统有时会故意设置 MMU 中的页表,以产生 DSI 异常,然后再进行异常处理。Linux PowerPC 会重点处理这一类 DSI 异常。
- 当处理器访问的地址空间跨越页边界,而这相邻的两个页面使用的端模式不匹配,即一个使用大端模式,一个使用小端模式时,E500 内核将会产生 DSI 异常。Linux PowerPC 一般不会产生这种类型的 DSI 异常。因为在 Linux PowerPC 中,其管理的页面一般来说,只能全部是大端的,或全部是小端的,不会使用混合页面。
- 一些操作 Cache 的指令,有时也会引起 DSI 异常,比如试图改变一些已经被锁定的 Cache 行。Linux PowerPC 不能恢复这类 DSI 异常。
- 使用“lwarx”和“stwcx.”指令对 Cache-inhibited 空间进行访问时,E500 内核也可以产生 DSI 异常。Linux PowerPC 不能恢复这类 DSI 异常。

Linux PowerPC 使用 DataStorage 函数处理 DSI 异常事件。DataStorage 函数会调用 data_access 函数进一步处理 DSI 异常事件。在 Linux PowerPC 中,除了 DSI 异常处理程序可以调用 DataAccess 函数外,还可以从 DTLB Miss 异常处理程序中调用 DataAccess 函数。

7.7.1 Linux PowerPC 对 DSI 异常的处理

E500 内核在进入 DSI 异常之前,自动保存和设置一些寄存器,以加速 Linux PowerPC 对这个异常的处理过程。

- SRR0 寄存器,保存引发 DSI 异常的指令有效地址,以便异常返回时进行现场恢复。
- SRR1 寄存器,保存引发 DSI 异常时 MSR 寄存器,以便异常返回时进行现场恢复。
- ESR 寄存器,保存引发 DSI 异常的条件及状态。
- MSR 寄存器,保留 MSR 寄存器的 CE,ME 和 DE 位,其余位清零。

● DEAR 寄存器,保存引发 DSI 异常的数据有效地址,即对哪一个数据进行读写操作时引发了 DSI 异常。

Linux PowerPC 使用 DataStorage 函数处理 DSI 异常。

```
SET _ IVOR(2,DataStorage);
```

Linux PowerPC 使用宏 SET _ IVOR,将 DataStorage 函数的低 16 位地址赋值给寄存器 IVOR2。当 E500 内核的 DSI 异常来临时,DataStorage 函数可以在 IVPR 和 IVOR2 指定的地址处执行,DataStorage 函数的源代码如下所示:

```
START _ EXCEPTION(DataStorage)
mtspr SPRN _ SPRG0, r10        /* Save some working registers */
mtspr SPRN _ SPRG1, r11
mtspr SPRN _ SPRG4W, r12
mtspr SPRN _ SPRG5W, r13
mfcr r11
mtspr SPRN _ SPRG7W, r11
```

DataStorage 函数首先使用 E500 内核的 SPR 寄存器,保存在该异常处理函数中使用的寄存器。

```
mfspr  r10, SPRN _ ESR
andis.  r10, r10, ESR _ ST@h
beq  2f
```

如果在 E500 内核中,ESR 寄存器的 ST 位为 1,则表示由存储器写操作引发 DSI 异常,否则跳转到标签 2f 处执行。ESR 寄存器记录了引发异常的状态位,与 ISI/DSI 异常相关的位有 ST 位(Store operation),DLK 位(Data Cache Locking),ILK 位(Instruction Cache Locking)和 BO 位(Byte-ordering Exception)。

DataStorage 函数只能修复部分由存储器写操作引发的 DSI 异常,其他原因引发的 DSI 异常需要由标签 2f 中的 data _ access 函数进行修复。

```
mfspr r10, SPRN _ DEAR    /* Get faulting address */

lis r11, TASK _ SIZE@h
ori r11, r11, TASK _ SIZE@l
cmplw 0, r10, r11
bge 2f
```

这段程序对 DEAR 寄存器进行检查。如果引发 DSI 异常的数据有效地址属于 Linux 内核空间,则跳转到标签 2f,由 data _ access 函数处理当前 DSI 异常。

```
3:
  mfspr r11,SPRN _ SPRG3
  lwz r11,PGDIR(r11)
4:
  FIND _ PTE
```

这段程序从 SPRG3 寄存器中获得当前进程描述符的 thread 参数，之后获得当前进程的 PGD 表基地址。随后调用 FIND _ PTE 函数，获得引发 DSI 异常的虚拟地址所对应的 PTE 表。FIND _ PTE 函数的详细说明见 7.1.2 节。

```
andi. r13, r11, _PAGE_RW|_PAGE_USER|_PAGE_HWWRITE
cmpwi 0, r13, _PAGE_RW|_PAGE_USER
bne 2f      /* Bail if not */
```

如果当前访问的地址属于用户空间，_ PAGE _ RW 位有效且_ PAGE _ HWWRITE 位不有效，此时将继续处理，否则跳转到标签 2f，由 data _ access 函数修复该 DSI 异常。

_ PAGE _ RW 位和_ PAGE _ HWWRITE 位的详细说明见 7.1.2 节。发生这种情况的主要原因是用户改变了进程的 PTE 表，而没有与 TLB0 中的 PTE 进行同步。DataStorage 函数的主体只能修复这种类型的 DSI 异常，其他 DSI 异常事件都需要 DataStorage 函数调用标签 2f 中的 data _ access 函数进行修复。

```
  ori r11, r11, _PAGE_DIRTY|_PAGE_ACCESSED|_PAGE_HWWRITE
  stw r11, PTE_FLAGS_OFFSET(r12) /* Update Linux page table */
...
  mfspr r12, SPRN_MAS3
  rlwimi r12, r11, 0, 20, 31
  mtspr SPRN_MAS3,r12
  tlbwe
```

这段函数将_ PAGE _ DIRTY 位存放到页表中，同时使用 tlbwe 指令同步在 TLB0 中的页表 Entry。

```
mfspr  r11, SPRN_SPRG7R
mtcr   r11
mfspr  r13, SPRN_SPRG5R
mfspr  r12, SPRN_SPRG4R
mfspr  r11, SPRN_SPRG1
mfspr  r10, SPRN_SPRG0
rfi      /* Force context change */
```

DataStorage 函数最后恢复工作现场，并使用 rfi 指令结束当前异常处理程序。

7.7.2 data _ access 函数

由 7.7.1 节所示，DataStorage 函数只能处理一小部分 DSI 异常事件，绝大多数的 DSI 异常事件需要 DataStorage 函数调用 data _ access 函数进行处理。data _ access 函数的源代码如下所示：

```
data_access:
  NORMAL_EXCEPTION_PROLOG
  mfspr r5,SPRN_ESR  /* Grab the ESR, save it, pass arg3 */
```

```
    stw r5,_ESR(r11)
    mfspr r4,SPRN_DEAR   /* Grab the DEAR, save it, pass arg2 */
    andis. r10,r5,(ESR_ILK|ESR_DLK)@h
    bne 1f
    EXC_XFER_EE_LITE(0x0300, handle_page_fault)
1:
    addi r3,r1,STACK_FRAME_OVERHEAD
    EXC_XFER_EE_LITE(0x0300, CacheLockingException)
```

data_access 函数首先使用宏 NORMAL_EXCEPTION_PROLOG,确定中断服务程序使用的堆栈空间,同时将异常处理程序中使用的通用寄存器和状态寄存器压入中断堆栈空间保存。宏 NORMAL_EXCEPTION_PROLOG 的详细说明见 6.4.1 节。

如果在 E500 内核中,ESR 寄存器的 ILK 或者 DSK 位为 1,则表示由 Cache 操作引发 DSI 异常。此时 data_access 函数跳转到标签 1f 处,调用 CacheLockingException 函数处理相应的 DSI 异常事件。否则将使用宏 EXC_XFER_EE_LITE 调用 handle_page_fault 函数,处理 DSI 异常事件。宏 EXC_XFER_EE_LITE 的详细说明见 6.7.2 节。

CacheLockingException 函数的源代码见 ./arch/powerpc/kernel/traps.c 文件,对此有兴趣的读者可自行阅读。本书将重点介绍 handle_page_fault 函数。

1. do_page_fault 函数的处理流程

do_page_fault 函数在处理 DSI 异常事件时,需要对引起 DSI 异常事件的 address 进行检查,之后再作出相应的处理。其处理流程如图 7-17 所示。

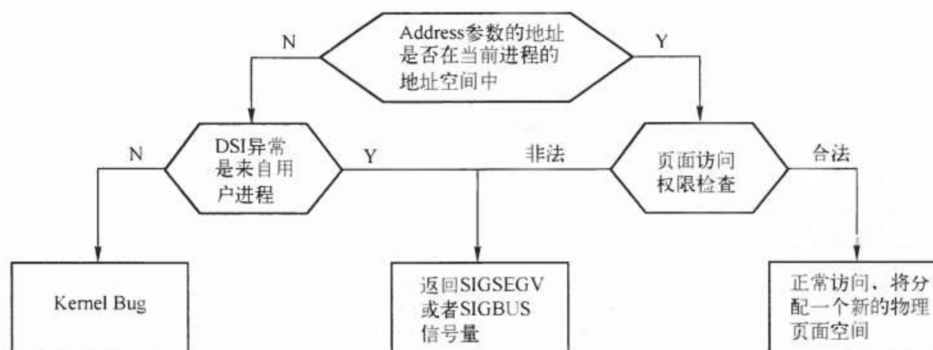


图 7-17　do_page_fault 函数的处理流程

如上图所示,do_page_fault 函数处理 DSI 异常时,有时返回 SIGSEGV 或者 SIGBUS 信号量。一般来说,SIGBUS(Bus error)意味着处理器总线不能正常访问 Address 地址处的内存,通常未对齐的数据访问或者硬件错误会导致这类情况。而 SIGSEGV(Segment fault)意味着 Address 地址是无效地址,没有物理内存与该地址对应。

在 Linux PowerPC 中,TLB Miss 异常中断处理程序有时也会跳转到 data_access 函数,从而执行 do_page_fault 函数。此时,do_page_fault 函数将会处理 TLB Miss 异常,并可能为引发 TLB Miss 异常的虚拟地址分配一个物理页面。

Linux PowerPC 的进程地址空间在创建时,并没有被加载到物理内存中。因此在进程的

运行过程中,会不可避免地发生当前进程的虚拟地址空间不在实际的物理内存中的情况,即当前进程的 pgd 表的 pte 表项无效。此时将引发 DTLB Miss 或 ITLB Miss 异常,然后跳转到 do_page_fault 处理函数中。do_page_fault 函数会重新分配一个物理页面,将进程地址空间分段加载。这种内存加载的方法也被称为 On-Demand 方法,采用这种方法的优点是不必为进程中的所有虚拟地址空间预留物理内存空间,从而有效避免了物理内存的浪费。

Linux PowerPC 还使用了写时拷贝 COW(Copy-on-Write)技术。COW 技术可以提高进程的创建速度及对物理内存的利用效率。Linux 较早的版本没有使用 COW 技术,进程创建时,子进程将复制父进程的地址空间,这将极大降低进程创建的速度,同时也会浪费一些物理内存空间。

在 Linux PowerPC 中,COW 技术的实现较为简单。在父进程创建子进程时,Linux 使用 COW 技术创建子进程的空间。在 Linux 系统中,子进程创建函数 do_fork 将调用 copy_process->copy_mm->dup_mm->dup_mmap->copy_page_range->...->copy_pte_range 函数建立 COW 页面。该函数进一步调用 copy_one_pte 函数实现 COW 页面。copy_one_pte 函数在 ./mm/memory.c 文件中,其中与 COW 页面相关的代码如下所示:

```
/*
 * If it's a COW mapping, write protect it both
 * in the parent and the child
 */
if (is_cow_mapping(vm_flags)) {
    ptep_set_wrprotect(src_mm, addr, src_pte);
    pte = pte_wrprotect(pte);
}
```

这段代码首先根据 vm_flags 判断,父进程创建子进程时,是否需要使用 COW 页面。如果需要,则将父子进程的 pte 表项设置为只读。此时父进程不需要为子进程创建地址空间,而是将父进程的 pte 表复制到子进程中。此后当父进程或者子进程对各自的进程空间进行写操作时,将产生 DTLB Miss 异常,并由 DTLB Miss 异常处理程序(DataAcess 函数)为子进程生成一个新的物理页面。

在 DataAcess 函数中,需要判断在进程地址空间中的数据页面是否为只读,如果是只读,则说明真正发生了向只读空间进行写入,否则认为是向 COW 页面进行写操作,此时将为 COW 页面重新分配一个物理页面,分离父子进程共用的物理页面,再将父子进程的 pte 表项都设置为可读写。采用 COW 技术可以有效提高子进程创建的速度和 Linux 内存管理子系统的效率。

2. do_page_fault 函数的代码分析

handle_page_fault 函数的源代码在 ./arch/powerpc/kernel/entry_32.S 文件中,该函数调用 do_page_fault 函数处理 DSI 异常事件,do_page_fault 函数在 ./arch/powerpc/mm/fault.c 文件中。

```
int __kprobes do_page_fault(struct pt_regs *regs, unsigned long address,
            unsigned long error_code)
```

do_page_fault 函数一共有三个参数,分别为 regs,address 和 error_code。

- regs 参数存放 PowerPC E500 内核的寄存器,handle_page_fault 函数使用 r3 寄存器传入此函数。
- address 参数存放 DEAR 寄存器,handle_page_fault 函数使用 r4 寄存器传入此函数。
- error_code 参数存放 ESR 寄存器,handle_page_fault 函数使用 r5 寄存器传入此函数。

do_page_fault 函数成功地将异常恢复后,将返回 0,否则将返回其他信号量,表示该函数不能正确恢复此 DSI 异常,需要使用信号机制来处理此异常。

在 Linux PowerPC 中,do_page_fault 函数的处理较为复杂。下文将分析该函数源代码的主要组成部分。

```
...
    /* On a kernel SLB miss we can only check for a valid exception entry */
    if (!user_mode(regs) && (address >= TASK_SIZE))
        return SIGSEGV;
```

在 IBM 的某些 64 位 PowerPC 处理器中,如 Power5 处理器,具有 SLB 寄存器(Segment Lookaside Buffer)。如果在内核空间的数据访问发生了 SLB Miss 事件,将最终进入 DSI 异常处理,此时 Linux PowerPC 直接将 SIGSEGV 返回。在 E500 内核中没有这类寄存器。

```
if (in_atomic() || mm == NULL) {
    if (!user_mode(regs))
        return SIGSEGV;
    /* in_atomic() in user mode is really bad,
       as is current->mm == NULL. */
    printk(KERN_EMERG "Page fault in user mode with"
        "in_atomic() = %d mm = %p\n", in_atomic(), mm);
    printk(KERN_EMERG "NIP = %lx MSR = %lx\n",
        regs->nip, regs->msr);
    die("Weird page fault", regs, SIGSEGV);
}
```

如果当前进程的 mm_struct 为空,则直接将 SIGSEGV 返回,核心进程的 mm_struct 为空。如果当前进程的 mm_struct 为空,且当前进程不是核心进程,则将进入 die 函数,整个 Linux 系统将被挂起,因为用户进程的 mm_struct 参数在正常情况下,不可能为空。

读者在阅读这段程序的时候需要注意,DSI 异常处理程序主要针对用户进程访问数据时引起的 DSI 异常。Linux 系统缺省认为 Linux 内核是值得信任的。因此对于 Linux 内核程序产生的错误,Linux 系统并不做恢复处理,只是保存错误状态并将系统挂起。此后 Linux 内核开发者去分析这些错误信息,并争取尽快改正这些系统 Bug。

```
...
    vma = find_vma(mm, address);
    if (!vma)
        goto bad_area;
    if (vma->vm_start <= address)
```

```
    goto good_area;
  if (!(vma->vm_flags & VM_GROWSDOWN))
    goto bad_area;

  if (address + 0x100000 < vma->vm_end) {
    /* get user regs even if this fault is in kernel mode */
    struct pt_regs *uregs = current->thread.regs;
    if (uregs == NULL)
      goto bad_area;

    if (address + 2048 < uregs->gpr[1]
      && (!user_mode(regs) || !store_updates_sp(regs)))
      goto bad_area;
  }
  if (expand_stack(vma, address))
    goto bad_area;
```

这段程序的执行流程如下：

- 获得 address 参数的所在的内存空间段 vm_area_struct。
- 如果 address 参数小于当前内存空间段的起始地址，即当前地址不是合法地址，则转移到 good_area，否则继续。
- 检查当前的内存空间段是否可以向前增长以覆盖 address 参数，如果当前的内存空间段可以向前增长，则继续，否则跳转到 bad_area。
- 判断当前 address 参数所指向的虚拟地址后是否有 1 MB 空间，以便扩展当前进程的栈段空间。如果可以扩展，则继续，否则跳转到 bad_area。
- 使用 expand_stack 函数扩展进程的栈段，如果该函数返回不成功则跳转到 bad_area，否则跳转到 good_area。

这段函数的流程如图 7-18 所示。

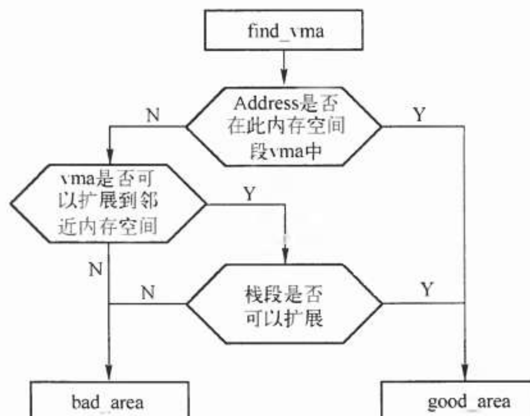


图 7-18　do_page_fault 函数处理流程

3. bad _ area

bad _ area 程序的代码如下所示：

```
bad_area:
  up_read(&mm->mmap_sem);

bad_area_nosemaphore:
  /* User mode accesses cause a SIGSEGV */
  if (user_mode(regs)) {
    _exception(SIGSEGV, regs, code, address);
    return 0;
  }

  if (is_exec && (error_code & DSISR_PROTFAULT)
    && printk_ratelimit())
    printk(KERN_CRIT "kernel tried to execute NX-protected"
         " page (%lx) - exploit attempt? (uid: %d\n",
         address, current->uid);
```

bad _ area 将对 Page Fault 异常处理函数的错误状态进行最后的处理。bad _ area 这段程序执行完毕后，可能发生以下几种情况：

（1）调用_ exception 函数将系统挂起。

（2）返回 0，SIGSEGV，SIGKILL 或者 SIGBUS。

4. good _ area

good _ area 这段程序首先对 address 地址进行检查。这里主要检查该地址的 pte 表项是否合理。如果 pte 表项不合理，则跳转到 bad _ area，否则执行 handle _ mm _ fault 函数。Linux PowerPC 认为以下两种 pte 表项是合理的：

（1）当前 address 参数指向的地址在程序空间段中没有对应的 pte 表项，或者 pte 表项没有被使用，此时 Page Fault 函数认为这是合理情况，handle _ mm _ fault 函数将重新为此地址申请一个页面空间。

（2）由数据写操作进入 Page Fault 异常处理函数，且当前 address 参数所在数据空间段可写，此时将申请新的物理页面。

以上这两种合理情况是按照 good _ area 程序的执行流程翻译过来的，因此读起来有些拗口。如果将这两种情况说得简单些，就是在这些情况下，do _ page _ fault 函数可以为导致该异常的虚拟地址申请新的页面。

Page Fault 异常处理函数使用 handle _ mm _ fault 函数进一步处理这两种情况，该函数的定义在 ./include/linux/mm.h 文件中，执行流程如图 7-19 所示。

handle _ mm _ fault 函数的处理较为复杂。其执行流程如下所示。

（1）此函数首先使用 pte _ alloc _ map 分配一个 pte 表项。

（2）然后调用 handle _ pte _ fault 函数，使用 Buddy System 进行页面分配。用户进程所使用的空间没有在实际的物理空间中时，如该进程第一次进行数据访问时，将使用 do _ no _ page

函数在 Buddy System 中申请一个物理页面;当用户进程所使用的空间需要与 SWAP 区进行交换时,将使用 do _ swap _ page 函数;当 Linux 系统处理 COW 页面时,将使用 do _ wp _ page 函数创建新的物理页面。

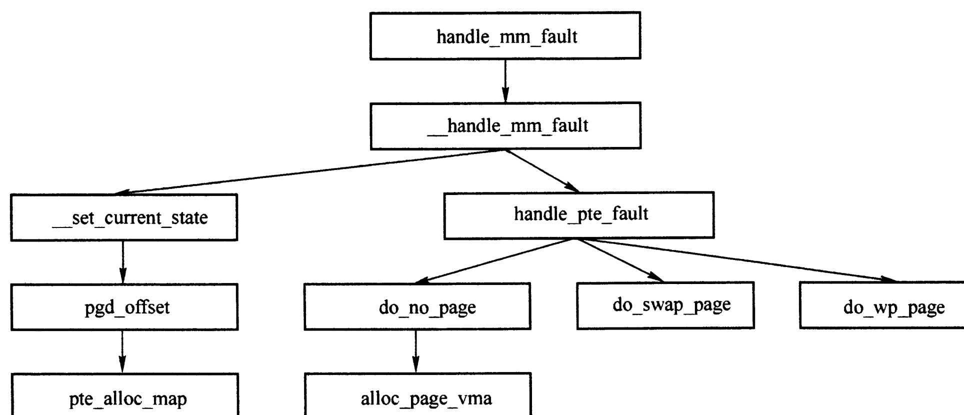


图 7-19 handle _ mm _ fault 函数处理流程

第8章 Linux PowerPC的初始化

在x86处理器系统和PowerPC处理器中,Linux系统的初始化过程非常类似。Linux系统的初始化是一个复杂的过程,Linux PowerPC的引导一般要分为以下三大步骤:

(1)使用Boot Loader程序,加载Linux PowerPC的内核文件到内存。

(2)Linux PowerPC的子系统初始化。

(3)对Linux PowerPC的应用程序初始化。

Linux PowerPC使用U-Boot或者Yaboot作为Boot Loader程序。Freescale的PowerPC处理器一般使用U-Boot,加载Linux PowerPC内核,而IBM的一些服务器主要使用Yaboot加载Linux PowerPC内核。

采用哪种Boot Loader程序并不重要,重要的是Boot Loader程序与Linux PowerPC之间如何进行信息传递,Boot Loader程序如何将Linux PowerPC的内核文件存放到内存中。Linux PowerPC系统的初始化如图8-1所示。

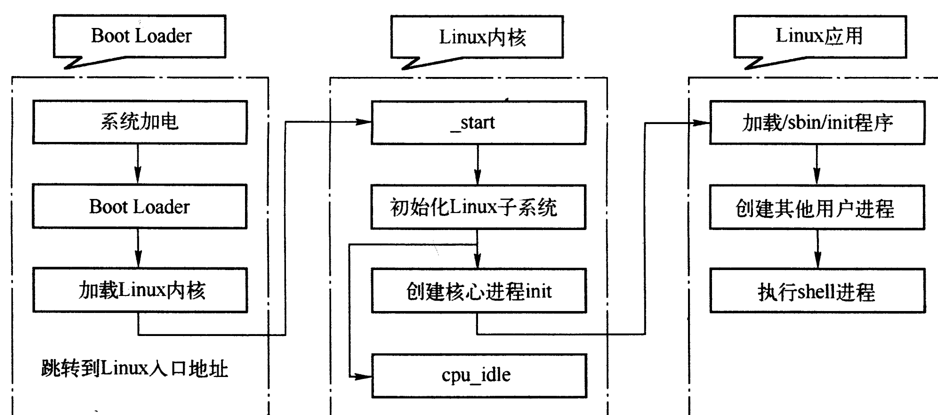


图8-1 Linux PowerPC初始化流程

Boot Loader程序负责处理器系统的基本初始化,并收集整个处理器系统的资源,如内存大小、处理器频率、外设的使用情况等一系列信息。这些信息由Boot Loader程序传递给Linux内核,随后Boot Loader程序将Linux PowerPC内核复制到物理内存0地址开始的物理内存中。基于E500内核的Linux PowerPC需要从物理地址0(其虚拟地址为0xC0000000)开始执行。

如果用户因为某些需要,使用基于E500内核的PowerPC处理器时,Boot Loader程序没有将Linux PowerPC内核复制到物理地址0中,程序员需要改动Linux PowerPC初始化部分的MMU映射。因为基于E500内核的Linux PowerPC认为Boot Loader将Linux PowerPC的入口地址设在物理内存0,并在此处开始执行。

有些Linux内核的维护者希望将Boot Loader程序与Linux PowerPC内核完全分离,以便维护这两段代码。但是目前这一想法还没有实现。将Boot Loader与Linux PowerPC内核有

机联系在一起,也许会更好。因为程序员很难清晰界定出 Boot Loader 程序与 Linux 内核之间的界限。

在 Boot Loader 完成对 Linux PowerPC 的加载后,处理器将从 Boot Loader 跳转到 Linux 内核中执行,对 Linux PowerPC 进行初始化。Linux PowerPC 的初始化共分为两部分,内核部分的初始化和核心进程 init 的执行。

内核部分的初始化将 Linux 的子系统初始化完毕后,创建核心进程 init。此后,核心进程 init 会继续执行。此时调度程序已经初始化完毕,并将在合适的时机执行 cpu_idle 函数,cpu_idle 函数是 idle 进程的程序体。在处理器没有活跃进程时,Linux 系统执行 idle 进程。在 Linux 系统中,该进程被称为进程 0。

核心进程 init 完成 Linux PowerPC 的各项配置后,将加载/sbin/init 程序初始化其他用户进程,加载/sbin/init 程序所产生的进程被称为进程 1,核心进程 init 也是进程 1。只是在核心进程 init 的执行后期,其进程地址空间被更换为加载/sbin/init 程序所产生的进程。随后,进程 1 根据/etc/inittab 文件创建其他的用户进程。一般来说,进程 1 会执行 login 进程进行登录操作,之后执行 Shell 进程,产生 Shell 界面,供用户与 Linux 系统进行交互。

本章主要介绍 Linux 子系统的初始化,即从 Boot Loader 加载内核到执行/sbin/init 程序之间的代码。这些代码共分为两大部分组成,第一部分从 Linux 系统程序入口地址_start 开始直到调用 start_kernel 函数,本书将此部分代码称为 Linux 系统的一次引导;而第二部分从 start_kernel 函数开始直到使用 kernel_thread 函数创建 init 进程,本书将此部分代码称为 Linux 系统的二次引导。

在介绍 Linux 系统的一次,二次引导之前,我们需要首先确定 Linux PowerPC 程序入口地址。在 ./arch/powerpc/Makefile 文件中定义了以 head-开始的一些参数,其代码如下所示:

```
head-y           := arch/powerpc/kernel/head_32.o
head-$(CONFIG_PPC64)          := arch/powerpc/kernel/head_64.o
head-$(CONFIG_8xx)            := arch/powerpc/kernel/head_8xx.o
head-$(CONFIG_4xx)            := arch/powerpc/kernel/head_4xx.o
head-$(CONFIG_44x)            := arch/powerpc/kernel/head_44x.o
head-$(CONFIG_FSL_BOOKE)         = arch/powerpc/kernel/head_fsl_booke.o

head-$(CONFIG_PPC64)          += arch/powerpc/kernel/entry_64.o
head-$(CONFIG_PPC_FPU)           += arch/powerpc/kernel/fpu.o
```

基于 E500 内核的 Linux PowerPC,其 CONFIG_FSL_BOOKE=y,其他配置都为 n,因此 head-y 参数等于 arch/powerpc/kernel/head_fsl_booke.o。由此可见基于 E500 内核的 Linux PowerPC 的系统引导从 head_fsl_booke.S 文件开始。

8.1 Open Firmware

最新的 U-Boot 源代码支持 OF(Open Firmware)结构,但是基于 PowerPC E500 内核的 Linux 2.6.20 源代码并不支持 OF 结构。其他的 PowerPC 处理器,如 604E 内核、603E 内核、E300 内核等,已经在 Linux 2.6.20 中支持了 OF 结构。因此本书将对 Open Firmware 进行简

单的介绍，目前 Linux 2.6.22 已经支持 E500 内核的 OF 结构。

Linux PowerPC 在 head _ fsl _ booke.S 文件的开始处，将 Boot Loader 程序传递的参数保存，其源代码如下所示：

```
mr      r31,r3
mr      r30,r4
mr      r29,r5
mr      r28,r6
mr      r27,r7
li      r24,0   /* CPU number */
```

在上述代码中，通用寄存器 r3，r4，r5，r6 和 r7 被保存到通用寄存器 r31，r30，r29，r28 与 r27 中。其中 r3～r7 寄存器的值是从 Boot Loader 程序中传递过来的。如果当前 Boot Loader 不支持 OF 结构，Boot Loader 程序将 Linux 系统复制到内存中时必须将以下数值放入 r3～r7 寄存器中：

- 通用寄存器 r3 存放 bd _ info 的地址指针，bd _ info 用来描述当前处理器系统的硬件信息，包括处理器频率、物理内存大小、网卡的地址等一系列信息。
- 通用寄存器 r4 存放 Init Ramdisk(initrd)的起始地址。
- 通用寄存器 r5 存放 Init Ramdisk(initrd)的结束地址。
- 通用寄存器 r6 存放内核的命令行参数的起始地址。使用 U-Boot 作为 Boot Loader，在引导 Linux PowerPC 内核时，会将 bootargs 环境变量传递给 Linux 系统，寄存器 r6 保存 bootargs 环境变量的起始地址。
- 通用寄存器 r7 存放内核的命令行参数的起始地址。

当采用 OF 结构进行 Linux 系统引导时，通用寄存器 r3～r7 传递的数值发生了一些变化，如表 8-1 所示。

表 8-1　U-Boot 传递给 Linux PowerPC 的参数

Linux 内核中的参数	不支持 OF 结构	支持 OF 结构
通用寄存器 r3	bd _ info 的地址	指向 OF Tree 结构的物理地址
通用寄存器 r4	initrd 的起始地址	指向 Linux 内核所在的物理地址
通用寄存器 r5	initrd 的结束地址	空
通用寄存器 r6	命令行参数起始地址	空
通用寄存器 r7	命令行参数结束地址	空

由上表所示，Boot Loader 支持 OF 结构时，仅使用通用寄存器 r3 和 r4 保存 OF 结构的信息和 Linux 内核所在的物理地址。在 E500 内核中，由于程序不能直接访问物理地址，在基于 E500 内核的 Linux PowerPC 中，使用通用寄存器 r4 保存 Linux 内核所在的有效地址。由上文可知，在 Linux PowerPC 初始化时，通用寄存器 r3 指向 OF Tree 结构的物理地址，这个 OF Tree 结构也被称作 dtb(Device Tree Block)。dtb 的组成如图 8-2 所示。

Boot Loader 加载 Linux 系统时，dtb 被放入指定的内存中，供 Linux 系统在引导过程中使用。如果 Linux PowerPC 支持 OF 结构，则调用 machine _ init ->early _ init _ devtree 函数，使

用 dtb 获得当前处理器系统的硬件信息。本书由于篇幅有限，仅对 Open Firmware 结构的基本知识作简要介绍，有关该结构的详细知识参见 ./Documentation/powerpc/booting-without-of.txt 文件。

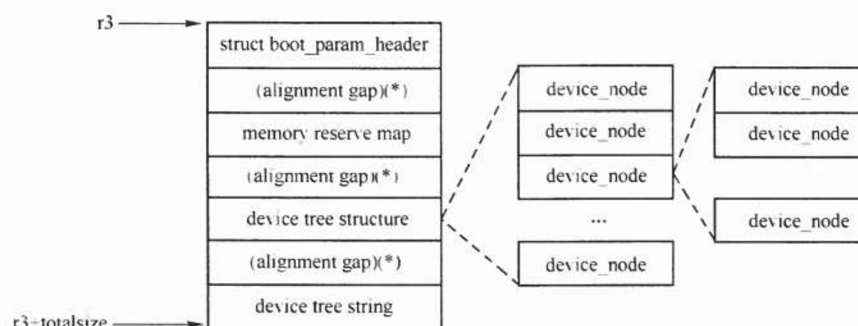


图 8-2　Device Tree Block 的组成

8.1.1　dtb 的数据结构

dtb 由 boot_param_header，device tree 结构和 device tree string 三大部分组成。

1. boot_param_header 结构

boot_param_header 结构在 ./arch/powerpc/boot/flatdevtree.h 文件中定义，用来保存 dtb 的头信息。

```
struct boot_param_header {
    u32 magic;                /* magic word OF_DT_HEADER */
    u32 totalsize;            /* total size of DT block */
    u32 off_dt_struct;        /* offset to structure */
    u32 off_dt_strings;       /* offset to strings */
    u32 off_mem_rsvmap;       /* offset to memory reserve map */
    u32 version;              /* format version */
    u32 last_comp_version;    /* last compatible version */
   /* version 2 fields below */
    u32 boot_cpuid_phys;      /* Physical CPU id we're booting on */
   /* version 3 fields below */
    u32 dt_strings_size;      /* size of the DT strings block */
};
```

- magic 参数保存 OF 结构的标志字。Linux 系统使用宏 OF_DT_HEADER (0xd00dfeed)表示该标志字。
- totalsize 参数保存 dtb 的大小。
- of_dt_struct 参数记录图 8-2 中 device tree 结构在 dtb 中的偏移。
- off_dt_strings 参数记录图 8-2 中 device tree string 在 dtb 中的偏移。
- off_mem_rsvmap 参数记录图 8-2 中 memory reserve map 在 dtb 中的偏移。Boot Loader 程序引导 Linux 系统的过程时，有些内存区域必须被预留出来，如存放 dtb 的内存区域；

如果 Linux 系统支持 initrd，还需要将 initrd 占用的内存区域预留出来。在 Linux 系统初始化时不能使用 off_mem_rsvmap 参数预留的区域。在 dtb 中，memory reserve map 表的每一个 Entry 由四个字节组成，前两个字节用来表示所预留空间的地址，而后两个字节用来表示所预留空间的大小。

- version 和 last_comp_version 参数记录 dtb 的版本号。
- boot_cpuid_phys 参数在 Linux SMP 中有意义。Linux SMP 将有 CPU 分为 BSP(Boot Stap Processor)和 AP(Application Processor)两类。其中 BSP 进行 Linux SMP 的初始化和引导其他 AP，AP 用作 Linux SMP 中的运算 CPU。boot_cpuid_phys 参数表示哪一个 CPU 用作 BSP。
- dt_strings_size 参数记录 device tree string 表的大小。

2. device_tree 结构

如图 8-2 所示在 dtb 中，device_tree 结构由 device node 组成，这些 device node 又可以由多个子 device node 组成。OF 使用 device_tree 结构对一个处理器系统进行描述。一个简单的 device_tree 结构如下所示。

```
/ o device-tree
      |- name = "device-tree"
      |- model = "MyBoardName"
      |- compatible = "MyBoardFamilyName"
      |- #address-cells = <2>
      |- #size-cells = <2>
      |- linux,phandle = <0>
      |
      o cpus
      |  |- name = "cpus"
      |  |- linux,phandle = <1>
      |  |- #address-cells = <1>
      |  |- #size-cells = <0>
      |  |
      |  o PowerPC,970@0
      |    |- name = "PowerPC,970"
      |    |- device_type = "cpu"
      |    |- reg = <0>
      |    |- clock-frequency = <5f5e1000>
      |    |- linux,boot-cpu
      |    |- linux,phandle = <2>
      |
      o memory@0
        |- name = "memory"
        |- device_type = "memory"
        |- reg = <00000000 00000000 00000000 20000000>
```

以上 device _ tree 结构分别描述处理器系统中使用的 CPU 和 Memory，该 device _ tree 包含一个根节点，每一个 device _ tree 中都含有唯一的根节点/。在根节点/中包含一个 cpus 和一个 memory 节点。而 cpus 节点中又包含一个子节点，“PowerPC, 970@0”节点。

从这个结构中可以发现，该处理器系统使用 IBM 的 PowerPC 970 作为主处理器，其系统频率为 1600000000 Hz(0x5f5e1000)。Memory 的大小为 512 MB，起始地址为 0。

下文将以 cpus 节点为例说明 device node 在内存中的存放规则。

- device node 使用宏 OF _ DT _ BEGIN _ NODE，其值为 0x0000 - 0001，作为 device node 的开始标志。
- 存放 device _ node 的节点名称，对于 cpus 节点，该值为“cpus”；对于 memory 节点，该值为“memory@0”，之后进行 4 字节对界。
- 存放节点的各个分项。首先使用 OF _ DT _ PROP 作为开始标志，然后存放节点中的各个分项。对于 cpus 节点，这些分项包括“linux, phandle”，“# address - cells”和“# size - cells”，其中每一个分项由名字和数值两部分组成。“name”分项不在 device node 中保存，而是保存到 dtb 的 device tree string 表中。
- 进行 4 字节对界。
- 如果一个 device node 具有子节点，则按照上述规则将子节点保存。在 cpus 节点中包含子节点“PowerPC, 970@0”。
- device node 使用宏 OF _ DT _ END _ NODE 作为节点的结束标志。

Linux 系统在进行初始化时将分析 dtb，然后将 dtb 中的 device node 存放到 device _ node 结构中，device _ node 结构的源代码如下所示：

```
struct device_node {
    const char * name;
    const char * type;
    phandlenode;
    phandle linux_phandle;
    char * full_name;

    struct  property * properties;
    struct  property * deadprops; /* removed properties */
    struct  device_node * parent;
    struct  device_node * child;
    struct  device_node * sibling;
    struct  device_node * next;   /* next device of same type */
    struct  device_node * allnext;   /* next in list of all nodes */
    struct  proc_dir_entry * pde;   /* this node's proc directory */
    struct  kref kref;
    unsigned long _flags;
    void * data;
};
```

Linux 系统将在 setup _ arch -> unflatten _ device _ tree 函数中扫描 dtb，将 dtb 中的节点信

息复制到 device _ node 结构中，最后使用 device _ node ->allnext 指针将所有 device _ node 结构组成一个单项链表，并使用 allnodes 指针保存这个链表头指针。unflatten _ device _ tree 函数在 ./arch/powerpc/kernel/prom.c 文件中。

Linux 系统还定义了一个特殊的 device _ node，即 of _ chosen，of _ chosen 描述当前处理器系统 dtb 中的 chosen 节点。chosen 节点包含 Linux 系统需要的环境变量、引导参数和 initrd 信息。在 Linux 系统中，首先使用 early _ init _ dt _ scan _ chosen 函数，从 dtb 的 chosen 节点中获得 bootargs、cmd _ line 等 Linux 系统引导参数，然后调用 unflatten _ device _ tree 函数将 chosen 节点的全部信息保存到全局变量 of _ chosen 中。

3. dtb 的生成

由上文可知，Linux 系统引导时需要 dtb 文件，但是 dtb 文件是一个二进制文件，不便于维护。为此 Open Firware 提供了 dtc(Device Tree Compile)编译器可以将文本文件 dts 转换为二进制文件 dtb，每一个处理器系统都有自己的 dts 文件。

Linux PowerPC 在 ./arch/powerpc/boot/dts 目录中存放了许多 dts 文件。MPC8541CDS 板的 dts 文件为 mpc8541cds.dts。

dtc 编译器的使用方法如下所示：

```
dtc [-I <input-format>] [-O <output-format>]
  [-o output-filename] [-V output _ version] input _ filename
```

input-format 可以使用以下三个参数：

- "dtb"。表示输入文件为 dtb 文件。
- "dts"。表示输入文件为 dts 文件。
- "fs"。表示输入文件与/proc/device - tree 文件的格式相同。

output-format 可以使用以下三个参数：

- "dtb"。表示输出文件为 dtb 文件。
- "dts"。表示输出文件为 dts 文件。
- "asm"。表示输出文件为汇编语言文件。

如果 output-format 为"dtb"时，output _ version 用来规定生成的 dtb 文件的版本号，目前 dtb 文件可用的版本号为 1,2,3 和 16，output-format 的缺省值为 3。input _ filename 和 output-filename 分别存放输入和输出文件名。从 dtc 编译器的使用方法中发现，dtc 编译器不仅可以实现 dtc 文件到 dtb 文件的转换，也可以实现 dtb 文件到 dtc 文件的转换。

8.1.2 Open Firmware 的 API 函数

OF 提供了许多 API，查找保存在全局链表 allnodes 中的 device node。这些 API 函数的源代码见 ./arch/powerpc/kernel/prom.c 文件。其主要 API 如下所示：

(1) of _ get _ flat _ dt _ root 函数。该函数用来查找在 dtb 中的根节点。

(2) of _ find _ node _ by _ path 函数。该函数根据 device _ node 结构的 full _ name 参数，在全局链表 allnodes 中，查找合适的 device _ node。该函数的使用方法如下所示：

```
struct device _ node * cpus;
cpus = of _ find _ node _ by _ path("/cpus");
```

(3) of _ find _ node _ by _ name 函数。该函数的使用方法如下所示：

```
struct device_node *np;
np = of_find_node_by_name(NULL, "firewire");
```

如果 of _ find _ node _ by _ name 函数的第一个参数为 NULL，该函数根据 device _ node 结构的 name 参数，在全局链表 allnodes 中查找合适的 device _ node。

(4) of _ find _ node _ by _ type 函数。该函数的使用方法如下所示：

```
struct device_node *tsi_pci;
tsi_pci = of_find_node_by_type(NULL, "pci");
```

如果 of _ find _ node _ by _ name 函数的第一个参数为 NULL，该函数根据 device _ node 结构的 type 参数，在全局链表 allnodes 中查找合适的 device _ node。

(5) of _ find _ node _ by _ phandle 函数。该函数根据 device _ node 结构的 linux _ phandle 参数，在全局链表 allnodes 中查找合适的 device _ node。该函数的使用方法如下所示：

```
struct device_node *phy;
phy = of_find_node_by_phandle(*ph);
```

上述这些函数根据 device _ node 结构的相应参数，寻找到合适的 device node 之后，可以使用以下函数寻找 device _ node 结构中相应的分项。

(6) of _ find _ property 函数。该函数根据 property 结构的 name 参数，在指定的 device node 中查找合适的 property。该函数的使用方法如下所示：

```
struct device_node *np;
const char *name;
int *lenp;
struct property *pp = of_find_property(np,name,lenp);
```

(7) of _ address _ to _ resource，of _ get _ address 函数。这两个函数对指定的 device node 结构进一步分解，以获得指定的分项。这两个函数在 ./arch/powerpc/kernel/prom _ parse.c 文件中定义。在该文件中还有一系列用来分解 pci 节点和 pic 节点的一些专用函数，其中 pci 节点与 PCI 总线的配置有关，而 pic 节点与 PIC 中断控制器有关。

8.2 Linux PowerPC 的一次引导

Linux PowerPC 在进入 start _ kernel 函数，进行一些与体系结构无关的初始化之前，需要做一些必要的准备工作。基于 E500 内核的处理器由于 MMU 的设置与其他 PowerPC 体系结构有很大的不同，因此其初始化过程也不尽相同。

基于 E500 内核的 PowerPC 处理器，MMU 无法关闭。因此 Boot Loader 程序必须要对 MMU 进行设置，然后将 Linux 内核下载到物理内存 0 处开始执行。由于 Linux PowerPC 并不知道 Boot Loader 程序如何对 MMU 的相应 TLB 进行配置，因此 Linux PowerPC 需要对 MMU 进行重新初始化。之后 Linux PowerPC 将对中断向量、内存系统、处理器系统进行最基本的初始化。其执行顺序如图 8-3 所示。

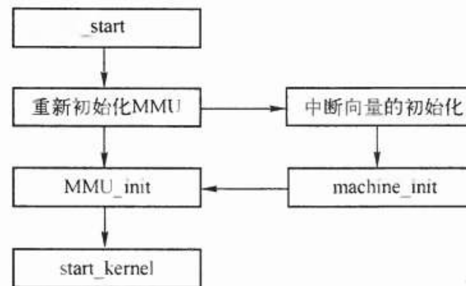


图 8-3 Linux PowerPC 的一次引导流程

8.2.1 MMU 的重新初始化

与 Linux PowerPC 的 MMU 管理相关的代码在 head_fsl_booke.S 文件中。这段代码由 Kumar Gala 编写，在某些方面仍然有些不足。也许在不远的将来会有人重新编写这段代码，但是这段程序仍然有闪光之处。

Linux PowerPC 在 Linux 内核的入口地址处，对 E500 内核的 MMU 重新进行配置。这段代码首先清除 Boot Loader 程序对 MMU 的设置，之后重新配置 MMU。这样做的主要目的是，尽可能地将 Boot Loader 程序与 Linux PowerPC 的代码独立，以便对各自的程序进行维护。采用这种方法，合理清除了 Boot Loader 程序对 MMU 设置。这段代码共有以下几个执行步骤：

(1) 首先使无效 TLB1 中的所有 Entry，仅保留当前程序正在使用的 Entry。

(2) 在 TLB1 中建立一个临时 Entry，然后让当前程序跳转到临时 Entry 的空间中。

(3) 清除程序最开始使用的 Entry。

(4) 将 TLB1 的 Entry0 初始化，之后让当前程序跳转到 Entry 0 的空间中。

(5) 最后清除临时 Entry。

MMU 重新初始化的源代码详解如下：

```
    bl invstr                  /* Find our address */
invstr: mflr r6                /* Make it accessible */
    mfmsr r7
    rlwinm r4,r7,27,31,31      /* extract MSR[IS] */
    mfspr r7, SPRN_PID0
    slwi r7,r7,16
    or r7,r7,r4
    mtspr SPRN_MAS6,r7
    tlbsx 0,r6                 /* search MSR[IS], SPID=PID0 */
```

- Linux 内核进行引导时，并不知道自己在哪个地址处执行，因此需要使用“bl”指令获得程序的运行地址，以便下文程序使用此地址进行 MMU 的配置。随后将此地址保留在通用寄存器 r6 中。“bl invstr”指令首先将当前指令的地址 CIA(Current Instruction Address)加 4，并将此结果存放到 LR 寄存器中，此时 LR 寄存器中将保留 CIA+4 的有效地址。

- 之后使用 PowerPC 著名的"rlwimn"指令，其主要作用是将 MSR 寄存器中的 IS 位单独取出，并放到通用寄存器 r4 的 63 位中。具体地说，该指令将 MSR 寄存器保存在通用寄存器 r7 中。将 r7 寄存器循环左移 27 位，然后与 0x1 进行与操作，并将结果存入通用寄存器 r4 中。
- 将当前寄存器 PID0 的值存入通用寄存器 r7 中，之后将其左移 16 位后与通用寄存器 r4 相与后将该值存入寄存器 MAS6。这段程序之所以需要将 MSR 寄存器和 PID0 寄存器进行移位的原因是 MAS6 寄存器的 SPID0 域在第 40～47 位，而 SAS 域在第 63 位。
- 使用 tlbsx 指令搜索通用寄存器 r6 所保存有效地址所在 TLB 的 Entry，并将结果存入寄存器 MAS0～3 中。

```
match_TLB:
  mfspr r7,SPRN_MAS0
  rlwinm r3,r7,16,20,31      /* Extract MAS0(Entry) */

  mfspr r7,SPRN_MAS1          /* Insure IPROT set */
  oris r7,r7,MAS1_IPROT@h
  mtspr SPRN_MAS1,r7
  tlbwe
```

这段代码使用 tlbsx 指令将从 TLB1 中获得的 Entry 保护起来，即将 MAS1 中的 IPROT 位置 1。此后，当用户使用 tlbivax 指令进行 TLB 使无效操作时，不能将 TLB1 中的这个 Entry 置为无效。

```
/* 2. Invalidate all entries except the entry we're executing in */
    mfspr r9,SPRN_TLB1CFG
    andi. r9,r9,0xfff
    li r6,0                   /* Set Entry counter to 0 */
1: lis r7,0x1000              /* Set MAS0(TLBSEL) = 1 */
   rlwimi r7,r6,16,4,15       /* Setup MAS0 = TLBSEL | ESEL(r6) */
   mtspr SPRN_MAS0,r7
   tlbre
   mfspr r7,SPRN_MAS1
   rlwinm r7,r7,0,2,31        /* Clear MAS1 Valid and IPROT */
   cmpw r3,r6
   beq skpinv                 /* Dont update the current execution TLB */
   mtspr SPRN_MAS1,r7
   tlbwe
   isync
skpinv: addi r6,r6,1          /* Increment */
  cmpw r6,r9                  /* Are we done? */
  bne 1b                      /* If not, repeat */
```

- 这段程序从 E500 内核的 TLB1CFG 寄存器中，获得 TLB1 一共有多少个 Entry，并将该值存入通用寄存器 r9。

- 这段程序认为 Boot Loader 程序使用了 TLB1 的 Entry0 描述 Linux 内核空间。因此将通用寄存器 r6 的值赋为 0。这种做法无可厚非，程序员编程时，总是要基于某种假设，很难做到将所有情况都处理干净，只是我希望在程序员做接口程序时，需要对使用的这些假设条件进行检查，尤其是对一个并不刻意追求效率的初始化程序。
- 之后检查 TLB1 的 Entry0～15。除了当前程序使用的 Entry 0 之外，以上程序使用 tlbwe 指令将 TLB1 其他的 Entry 中的 V 位和 IPROT 位清零，即使无效其他 Entry。该段代码的实现较为简单，此处不再叙述。

```
...
/* 3. Setup a temp mapping and jump to it */
    andi. r5, r3, 0x1    /* Find an entry not used and is non-zero */
    addi r5, r5, 0x1
    lis r7,0x1000 /* Set MAS0(TLBSEL) = 1 */
    rlwimi r7,r3,16,4,15 /* Setup MAS0 = TLBSEL | ESEL(r3) */
    mtspr SPRN_MAS0,r7
    tlbre
```

- 将寄存器 r7 的值赋值到寄存器 MAS0 中。之前通用寄存器 r3 中保存了当前程序正在使用的 Entry 号，然后将此值加 1 赋值到通用寄存器 r5 中。
- 使用 tlbre 指令，根据寄存器 MAS0 的值，将 TLB1 的相应 Entry 存入寄存器 MAS1～3。

```
/* Just modify the entry ID and EPN for the temp mapping */
lis r7,0x1000              /* Set MAS0(TLBSEL) = 1 */
rlwimi r7,r5,16,4,15       /* Setup MAS0 = TLBSEL | ESEL(r5) */
mtspr SPRN_MAS0,r7
xori r6,r4,1               /* Setup TMP mapping in the other Address space */
slwi r6,r6,12
oris r6,r6,(MAS1_VALID|MAS1_IPROT)@h
ori r6,r6,(MAS1_TSIZE(BOOKE_PAGESZ_4K))@l
mtspr SPRN_MAS1,r6
mfspr r6,SPRN_MAS2
li r7,0                    /* temp EPN = 0 */
rlwimi r7,r6,0,20,31
mtspr SPRN_MAS2,r7
tlbwe
```

- 这段程序在 TLB1 中设置一个临时的 Entry。将寄存器 MAS0 的 TLBSEL 字段赋值为 1(使用 TLB1)，将 ESEL 字段设置为通用寄存器 r5 中的数值。
- 将寄存器 MAS1 的 V 位，IPROT 位和 TSIZE 字段分别设置为 1，1 和 4KB。并将寄存器 MAS2 的 EPN 字段设置为 0。
- 寄存器 MAS0～MAS3 的其他字段使用上一段程序 tlbre 指令读出的值。
- 提醒读者注意，这段程序仅为有效地址空间 0～4 KB 之间的区域建立临时 Entry 的映射。如果用户为 Linux 内核准备的有效地址空间不是以 0 开始的，下文的 rfi 指令有可

能会执行错误，可能会跳转到 ITLB Miss 异常处理程序中。此时 ITLB Miss 异常处理程序还没有被初始化，因此 rfi 指令还可能跳转到 Program 异常处理程序中。这段代码最致命的问题是，假定 Boot Loader 引导 Linux 内核时有许多前提条件，但是并没有对这些条件进行检查。

```
    xori r6,r4,1
    slwi r6,r6,5                /* setup new context with other address space */
    bl 1f                       /* Find our address */
1: mflr r9
    rlwimi r7,r9,0,20,31
    addi r7,r7,24
    mtspr SPRN_SRR0,r7
    mtspr SPRN_SRR1,r6
    rfi
/* 4. Clear out PIDs & Search info */
    li r6,0
```

- 使用 bl 指令获得当前 CIA + 4 的有效地址，然后将这一地址的第 52～63 位保存到通用寄存器 r7 中，然后加上 24，即 6 条指令的大小，此时 r7 保存的地址为 1: + 24，即指令“li r6,0”所在的有效地址。此时使用 rfi 指令后，程序将跳转到 TLB1 临时 Entry 所指向的“li r6,0”指令。
- 提醒读者注意，使用 rfi 指令进行程序跳转的好处是，程序跳转后将自动执行 isync 指令，以保证指令空间的同步，在 Linux PowerPC 的初始化阶段，使用 rfi 指令进行程序跳转比较常见。这里的 rfi 指令与中断返回没有任何关系。

至此，Linux 引导程序将运行在临时 Entry 所指定的空间中。

```
    mtspr SPRN_PID0,r6
#ifndef CONFIG_E200
    mtspr SPRN_PID1,r6
    mtspr SPRN_PID2,r6
#endif
    mtspr SPRN_MAS6,r6
```

这段程序将 E500 内核的寄存器 PID0～2 及寄存器 MAS6 置为 0。Linux PowerPC 没有使用 E500 内核的地址空间 1，也没有使用 PID 寄存器。

```
/* 5. Invalidate mapping we started in */
    lis r7,0x1000 /* Set MAS0(TLBSEL) = 1 */
    rlwimi r7,r3,16,4,15 /* Setup MAS0 = TLBSEL | ESEL(r3) */
    mtspr SPRN_MAS0,r7
    tlbre
    li r6,0
    mtspr   SPRN_MAS1,r6
```

```
    tlbwe
    /* Invalidate TLB1 */
    li  r9,0x0c
    tlbivax 0,r9
#ifdef CONFIG_SMP
    tlbsync
#endif
    msync
```

这段代码将最初存放 Linux 内核的 TLB1 的对应 Entry 使无效,此时,在当前系统中只有 TLB1 的临时 Entry 有效。

```
/* 6. Setup KERNELBASE mapping in TLB1[0] */
    lis r6,0x1000 /* Set MAS0(TLBSEL) = TLB1(1), ESEL = 0 */
    mtspr SPRN_MAS0,r6
    lis r6,(MAS1_VALID|MAS1_IPROT)@h
    ori  r6,r6,(MAS1_TSIZE(BOOKE_PAGESZ_16M))@l
    mtspr  SPRN_MAS1,r6
    li  r7,0
    lis  r6,KERNELBASE@h
    ori  r6,r6,KERNELBASE@l
    rlwimi  r6,r7,0,20,31
    mtspr  SPRN_MAS2,r6
    li  r7,(MAS3_SX|MAS3_SW|MAS3_SR)
    mtspr  SPRN_MAS3,r7
    tlbwe
```

这段程序初始化 TLB1 的 Entry0,此时 MAS 寄存器的各个字段的值如下:

- TLBSEL 字段为 01,即使用 TLB1。
- ESEL 字段为 0,使用 Entry 0。
- V 位为 IPROT,表示将使能并保护 TLB1 的 Entry0。
- TID 位为 0,表示使用全局 PID 寄存器,及所有的 PID 寄存器进行地址映射。
- TS 位为 0,表示所描述的 Entry0 将使用 E500 内核的地址空间 0,Linux PowerPC 只使用 E500 内核的地址空间 0。
- TSIZE 位为 BOOKE_PAGESZ_16M,即 16 MB。
- EPN 字段为 KERNELBASE,即 0xC0000000。
- WIMGE 位都为 0。
- RPN 字段为 0,SX、SW、SR 位为 1,表示超级用户可以对此 Entry 所描述的空间进行读写和执行操作。
- 使用 tlbwe 指令将 KERNELBASE~KERNELBASE+16M 这段的有效地址空间映射到 0~16 MB 物理地址空间中。至此 Linux PowerPC 完成了对 TLB1 中 Entry 0 的设置。

```
/* 7. Jump to KERNELBASE mapping */
    lis r7,MSR_KERNEL@h
    ori  r7,r7,MSR_KERNEL@l
    bl 1f            /* Find our address */
1: mflr r9
   rlwimi r6,r9,0,20,31
   addi  r6,r6,24
   mtspr  SPRN_SRR0,r6
   mtspr  SPRN_SRR1,r7
   rfi              /* start execution out of TLB1[0] entry */
```

- 将 SRR0,SRR1 寄存器分别赋为 MSR_KERNEL,和当前程序段 rfi 指令的下一条指令的有效地址。
- 使用 rfi 指令进行长跳转。此后指令将在 TLB1 的 Entry0 所描述的地址空间内执行。

```
/* 8. Clear out the temp mapping */
    lis r7,0x1000 /* Set MAS0(TLBSEL) = 1 */
    rlwimi r7,r5,16,4,15 /* Setup MAS0 = TLBSEL | ESEL(r5) */
    mtspr SPRN_MAS0,r7
    tlbre
    mtspr SPRN_MAS1,r8
    tlbwe
    /* Invalidate TLB1 */
    li  r9,0x0c
    tlbivax 0,r9
#ifdef CONFIG_SMP
   tlbsync
#endif
   msync
```

这段代码使无效 TLB1 的临时 Entry。至此,Linux 系统中只有 TLB1 的 Entry0 所描述的空间有效,其他地址空间都没有被描述。采用这种方法的目的是清除从 Boot Loader 程序中带来的对 TLB1 相应 Entry 的一些设置。

8.2.2 中断向量的初始化

E500 内核中断向量的初始化,即对 E500 内核的 IVPR 和 IVOR 寄存器赋值。Linux PowerPC 使用以下程序初始化 E500 内核的中断向量:

```
SET_IVOR(0, CriticalInput);
SET_IVOR(1, MachineCheck);
SET_IVOR(2, DataStorage);
SET_IVOR(3, InstructionStorage);
SET_IVOR(4, ExternalInput);
```

```
    SET_IVOR(5, Alignment);
    SET_IVOR(6, Program);
    SET_IVOR(7, FloatingPointUnavailable);
    SET_IVOR(8, SystemCall);
    SET_IVOR(9, AuxillaryProcessorUnavailable);
    SET_IVOR(10, Decrementer);
    SET_IVOR(11, FixedIntervalTimer);
    SET_IVOR(12, WatchdogTimer);
    SET_IVOR(13, DataTLBError);
    SET_IVOR(14, InstructionTLBError);
    SET_IVOR(15, Debug);
    SET_IVOR(32, SPEUnavailable);
    SET_IVOR(33, SPEFloatingPointData);
    SET_IVOR(34, SPEFloatingPointRound);
  #ifndef CONFIG_E200
    SET_IVOR(35, PerformanceMonitor);
  #endif
```

这段程序使用宏 SET_IVOR,将中断向量处理函数的地址存放到 IVOR 寄存器中,比如将 CriticalInput 函数的地址存放到 IVOR0 寄存器中。宏 SET_IVOR 的详细说明见 6.1.3 节。基于 E500 内核的 Linux PowerPC 需要处理的中断与异常由以上程序所示。本书中对一些关键的中断与异常如系统时钟异常、TLB Miss 异常、外部中断处理和 DSI 异常进行了详细介绍,其余的异常处理程序需要读者自行阅读。

```
/* Establish the interrupt vector base */
lis r4,interrupt_base@h /* IVPR only uses the high 16-bits */
mtspr   SPRN_IVPR,r4
```

以上程序初始化 IVPR 寄存器。

8.2.3 初始化进程 0

Linux PowrPC 在完成 MMU 的重新初始化和中断向量的初始化后,将创建进程 0,并由进程 0 完成 Linux PowerPC 的一次引导和二次引导。

进程 0 的运行将贯穿 Linux PowerPC 初始化过程。进程 0 开始运行前,Linux PowerPC 需要为其创建进程描述符与数据栈段,进程 0 使用的正文段空间与 Linux 内核使用的正文段空间相同。

```
/* ptr to current */
lis r2,init_task@h
ori r2,r2,init_task@l
```

init_task 是进程 0 使用的进程描述符,也是 Linux 系统中第一个进程描述符。该进程描述符的定义在 ./arch/powerpc/kernel/init_task.c 文件中。

init_task 描述符使用宏 INIT_TASK 在编译 Linux 系统时,对 init_task 进程描述符进

行静态赋值。宏 INIT _ TASK 的定义在 ./include/linux/init _ task.h 文件中。读者可以参考本书 5.1 节,分析 init _ task 进程描述符的初始化属性。

随后这段代码将进程描述符 init _ task 赋给通用寄存器 r2,由上文所示,通用寄存器 r2 保存当前正在运行的进程描述符。在 Linux 系统进行初始化时,当前运行的进程是进程 0,此时 Linux 系统使用通用寄存器 r2 保存 init _ task 的地址。

```
/* ptr to current thread */
addi r4,r2,THREAD /* init task's THREAD */
mtspr SPRN_SPRG3,r4
```

这段代码使用的宏 THREAD 与 offsetof(struct task _ struct, thread) 等效,即保存进程描述符 task _ struct->thread 参数在当前 task _ struct 结构中的偏移。

这段代码的主要目的是将专用寄存器 SPRG3 赋值为 r2->thread,即使用 SPRG3 寄存器保存当前进程描述符 thread 参数的地址。针对不同的 PowerPC 处理器,Linux PowerPC 对专用寄存器 SPRG3 的使用并不相同。基于 E500 内核的 PowerPC 处理器使用寄存器 SPRG3 保存 r2->thread 的有效地址,而基于 603E、604E 和 E300 内核的 PowerPC 处理器使用寄存器 SPR3 保存 r2->thread 的物理地址。

寄存器 SPRG3 是 Linux PowerPC 为基于 603E、604E 和 E300 等 PowerPC 内核设置的。这一类处理器进入中断或者异常处理程序时,MMU 会被自动关闭,此时 Linux PowerPC 必须使用 SPRG3 寄存器才能找到当前的进程描述符;基于 E500 内核的 PowerPC 处理器在进入中断或者异常处理模式时并不关闭 MMU,因此不需要在 SPRG3 寄存器中保存 r2->thread 的物理地址也可以找到当前进程的描述符。但是基于 E500 内核的 Linux PowerPC 依然使用 SPRG3 寄存器保存 r2->thread 的有效地址以便和其他体系的 Linux PowerPC 兼容。

进程 0 使用变量 init _ thread _ union 中的空间作为该进程的堆栈空间,同时该进程的 thread _ info 参数也使用此空间。与 init _ thread _ union 变量有关的源代码如下所示:

```
/* ./arch/powerpc/kernel/init_task.c */
union thread_union init_thread_union
  __attribute__((__section__(".data.init_task"))) =
    { INIT_THREAD_INFO(init_task) };

/* ./include/asm-powerpc/thread_info.h */
#define init_thread_info (init_thread_union.thread_info)
#define init_stack (init_thread_union.stack)

/* ./include/linux/sched.h */
union thread_union {
  struct thread_info thread_info;
  unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

读者可以由以上代码发现,变量 init _ thread _ union 在 ./arch/powerpc/kernel/init _ task.c 文件中定义,并放入 .data.init _ task 段中,init _ thread _ union 是 thread _ union 联合结

构的变量。在 thread _ union 结构中包含两个参数 thread _ info 和 stack，其中 thread _ info 和 stack 参数共享一段 8 KB 的空间。

Linux 系统在 ./include/asm - powerpc/thread _ info.h 文件中定义了两个宏，init _ thread _ info 和 init _ stack，分别用来访问进程 0 的 thread _ info 参数和堆栈。

```
/* stack */
lis r1,init_thread_union@h
ori r1,r1,init_thread_union@l
li r0,0
stwu r0,THREAD_SIZE-STACK_FRAME_OVERHEAD(r1)
```

这段程序为进程 0 建立堆栈空间。基于 E500 内核的 Linux PowerPC，THREAD _ SIZE 的值为 8 KB，STACK _ FRAME _ OVERHEAD 的值为一个栈帧的最小值 16。该程序首先使用 stwu 指令将 0 存放到前一个栈帧的底部，然后将通用寄存器 r1 指向当前栈帧的底部，以确定 init _ task 的堆栈结构。这段程序执行完毕后，init _ task 进程的进程描述符、正文段、数据段和堆栈段建立完毕，该进程程序运行空间已经基本建立完毕。

之后，初始化进程 init _ task 将依次调用 early _ init 函数、machine _ init 函数、MMU _ init 函数和 start _ kernel 函数，完成 Linux PowerPC 的初始化。

8.2.4 early _ init 函数

```
bl early_init
```

early _ init 函数的主要功能是判断当前处理器系统使用的内核类型，并作出相应的初始化操作。early _ init 函数在 ./arch/powerpc/kernel/setup _ 32.c 文件中定义，其源代码详解如下：

```
/* early_init 函数源代码片段 1 */
__init
unsigned long
early_init(int r3, int r4, int r5)
{
```

early _ init 函数共有三个输入参数和一个返回值。其中这些输入参数分别保存在通用寄存器 r3，r4 和 r5 中，基于 E500 内核的 Linux PowerPC 在 early _ init 函数中，没有使用这些输入参数。early _ init 函数的返回值为 Linux 内核运行的物理地址，该函数的返回值对于基于 E500 内核的 Linux PowerPC 没有意义。

```
/* early_init 函数源代码片段 2 */
    unsigned long phys;
    unsigned long offset = reloc_offset();
    struct cpu_spec *spec;

    /* Default */
    phys = offset + KERNELBASE;
```

这段函数使用 reloc _ offset 函数，计算 Linux 内核的连接地址和 Linux 内核实际运行地址之间的偏移，随后将这个偏移和 KERNELBASE 相加后赋给变量 phys。early _ init 函数执行结束后将 phys 变量返回。

变量 spec 是 cpu _ spec 类型的参数，cpu _ spec 类型在 ./include/asm-powerpc/cputable.h 中定义。该类型的主要数据成员如下：

- pvr _ value 参数存放 CPU 内核版本号。
- icache _ bsize，dcache _ bsize 参数存放 CPU 的指令和数据 Cache 行长度。
- cpu _ setup 参数指向一个初始化函数，该函数可以被 early _ init 函数调用，用户可以将一些需要在 Linux 系统启动的最初阶段执行的代码放入此函数中。目前基于 E500 内核的 Linux PowerPC 都没有使用这个参数。
- cpu _ name 参数保存当前处理器内核的名称，如 MPC8541 处理器的 cpu _ name 参数为"e500"。
- platform 参数保存当前系统使用处理器的名称，如 MPC8541 处理器的 platform 参数为"ppc8540"。在 Linux PowerPC 中，MPC8541/8555/8540/8560 的 platform 参数都为"ppc8540"。

```
/* early _ init 函数源代码片段 3 */
  /* First zero the BSS -- use memset _ io, some platforms don't have caches on yet */
  memset _ io((void __ iomem *)PTRRELOC(& __ bss _ start), 0, _ end - __ bss _ start);
```

这段程序使用 memset _ io 函数将 Linux 内核的 bss 段清零，bss 段存放未初始化的全局变量。这段程序使用函数 memset _ io 将以 PTRRELOC(& __ bss _ start)开始的数据段清零，这段数据段的大小为 _ end-__ bss _ start。__ bss _ start、_ end 及 __ bss _ start 变量来自 vmlinux.lds 文件，Linux 内核编译完成后这些变量将被初始化。

在 Linux 内核运行时，不能够直接使用这些编译值，因为 Boot Loader 不一定将 Linux 内核映像根据其编译地址存放到内存中，因此 Linux 系统需要使用宏 PTRRELOC，将程序的编译地址转换为实际的有效地址。宏 PTRRELOC 调用函数 add _ reloc _ offset，计算编译地址与实际的有效地址的差值，此函数的源代码在 ./arch/powerpc/kernel/misc.S 文件中。这段代码如下所示：

```
_GLOBAL(add _ reloc _ offset)
  mflr r0
  bl 1f
1: mflr r5
   LOAD _ REG _ IMMEDIATE(r4,1b)
   subf r5,r4,r5
   add r3,r3,r5
   mtlr r0
   blr
```

- 这段程序首先将 LR 寄存器保留在通用寄存器 r0 中。之后使用"bl 1f"指令获得 LR 寄存器的值，即"1: mflr r5"指令的有效地址。

- 将该指令的有效地址存放在通用寄存器 r5 中。
- 使用宏 LOAD_REG_IMMEDIATE 计算“1b”，即“1: mflr r5”指令的编译地址，并将其结果存入通用寄存器 r4 中。
- 将通用寄存器 r4 减 r5 的值赋予通用寄存器 r5。此后，通用寄存器 r5 存放 Linux 内核被 Boot Loader 加载后的有效地址减去其编译地址的差值，即有效地址与编译地址之间的偏移。
- 将通用寄存器 r3 加上 r5 的值，并赋给 r3，完成编译地址到有效地址的转换。最后将保存在通用寄存器 r0 中的 LR 寄存器还原，然后使用 blr 指令进行程序返回。

early_init 函数将 bss 段初始化完毕后，将使用以下代码获得当前处理器的类型。

```
/* early_init 函数源代码片段 4 */
  /*
   * Identify the CPU type and fix up code sections
   * that depend on which cpu we have.
   */
  spec = identify_cpu(offset, mfspr(SPRN_PVR));
```

Linux PowerPC 将所有 PowerPC 处理器系统的 cpu_spec 结构统一存放到全局数组 cpu_specs 中。Linux PowerPC 初始化时，early_init 函数根据当前处理器的 PVR 寄存器扫描全局数组 cpu_specs，以获得当前处理器系统使用的 cpu_spec 结构。

PVR 寄存器用来识别处理器系统使用的 PowerPC 内核，如 MPC8541 处理器使用 E500 内核，MPC8548 使用 E500 V2 内核，而 MPC8272 使用 603E 内核。对于 MPC85XX 系列的处理器，其 cpu_spec 参数初始化如下所示：

```
{ /* e500 */
  .pvr_mask       = 0xffff0000,
  .pvr_value      = 0x80200000,
  .cpu_name       = "e500",
  /* xxx - galak: add CPU_FTR_MAYBE_CAN_DOZE */
  .cpu_features   = CPU_FTRS_E500,
  .cpu_user_features  = COMMON_USER_BOOKE |
    PPC_FEATURE_SPE_COMP |
    PPC_FEATURE_HAS_EFP_SINGLE,
  .icache_bsize   = 32,
  .dcache_bsize     = 32,
  .num_pmcs         = 4,
  .oprofile_cpu_type   = "ppc/e500",
  .oprofile_type    = PPC_OPROFILE_BOOKE,
  .platform     = "ppc8540",
},
```

identify_cpu 函数根据 PVR 寄存器，获得当前处理器系统的 cpu_spec 参数。基于 E500 V1 内核的处理器，其 PVR 寄存器为 0x8020xxxx，该值与存放在 cpu_specs 数组的 e500 内核

使用的 cpu _ spec ->prv _ value 参数相同,因此 identify _ cpu 函数返回时将获得此 cpu _ spec。

early _ init 函数执行 identify _ cpu 函数后将调用 do _ feature _ fixups 函数。

```
/* early_init 函数源代码片段 5 */
  do_feature_fixups(spec->cpu_features,
      PTRRELOC(&__start___ftr_fixup),
      PTRRELOC(&__stop___ftr_fixup));

  return KERNELBASE + offset;
} /* End early_init */
```

do _ feature _ fixups 函数的源代码在 ./arch/powerpc/kernel/cputable.c 文件中,该函数使用了一个小技巧,以实现代码复用。

不同的 PowerPC 处理器具有不同的特性,如指令 Cache 和数据 Cache 的大小不同;指令集并不完全相同(有些指令在 E500 中存在而在 603E 内核中不存在,如指令 wrtee);不同的 MMU 管理策略;不同的电源管理方法等等。虽然这些 PowerPC 处理器之间有许多差异,但是这些处理器之间还是有非常多的共性。因此如果专门为这些 PowerPC 处理器提供不同的源文件显得过于冗余。为此,Linux PowerPC 处理与 PowerPC 处理器特性有关的代码时,使用宏 BEGIN _ FTR _ SECTION 和 END _ FTR _ SECTION _ IFSET 将这些代码放入 Linux 内核的单独的一个段__ ftr _ fixup 中。如下所示:

```
BEGIN_FTR_SECTION
  LOAD_BAT(4,r3,r4,r5)
  LOAD_BAT(5,r3,r4,r5)
  LOAD_BAT(6,r3,r4,r5)
  LOAD_BAT(7,r3,r4,r5)
END_FTR_SECTION_IFSET(CPU_FTR_HAS_HIGH_BATS)
```

首先宏 BEGIN _ FTR _ SECTION 和 END _ FTR _ SECTION _ IFSET 之间的源代码将被链接器放入到 Linux PowerPC 内核的__ ftr _ fixup 段中。identify _ cpu 函数确定 CPU 的 cpu _ spec 参数之后,根据 cpu _ spec ->cpu _ user _ features 参数,调用 do _ feature _ fixups 函数对__ ftr _ fixup 段进行修复。

- 如果一个 CPU 具有某特性,则操作该特性的代码将不作任何处理,如在上述程序中,如果当前处理器的 CPU 具有 CPU _ FTR _ HAS _ HIGH _ BATS 特性,do _ feature _ fixups 函数将不会对存放这段程序的__ ftr _ fixup 段进行改写。
- 如果 CPU 不具备该特性,则使用 do _ feature _ fixups 函数将操作该特性的代码全部替换成 nop 指令。do _ feature _ fixups 函数有一个立即数“0x60000000u”,该立即数与 nop 指令机器码相同。在 E500 内核中,nop 指令等效于“ori 0, 0, 0”指令。

提醒读者注意,在 PowerPC 处理器中,如果程序员对指令所在的内存进行修改后,必须清除指令和数据 Cache 中的对应 Cache 行,否则很可能造成指令、数据 Cache 及存储器内容不一致,从而导致错误。在 PowerPC 中,更改指令段的代码被称为 Self - Modify 代码。程序员在编写这类的代码时需要十分谨慎,在这类代码后,需要使用指令同步和存储同步指令。本书不会对 do _ feature _ fixups 函数进行逐行分析,但是希望读者能够准确理解该函数的含义。

8.2.5 machine _ init 函数

```
        mfspr r3,SPRN_TLB1CFG
        andi. r3,r3,0xfff
        lis r4,num_tlbcam_entries@ha
        stw   r3,num_tlbcam_entries@l(r4)

    /* Decide what sort of machine this is and initialize the MMU. */
        mr r3,r31
        mr   r4,r30
        mr   r5,r29
        mr   r6,r28
        mr   r7,r27
        bl machine_init
```

Linux PowerPC 从 early _ init 函数返回后,将从 TLB1CFG 寄存器中,获得当前 E500 内核的 TLB1 有多少个 Entry,然后将该结果存入全局变量 num _ tlbcam _ entries 中,之后调用 machine _ init 函数。machine _ init 函数在 ./arch/powerpc/kernel/setup _ 32.c 文件中,machine _ init 函数的主要功能有两个。

(1) 分析 OF Tree 结构,获得当前处理器的内存使用情况,创建 LMB 结构,同时获得当前处理器系统在 OF Tree 结构中的其他硬件信息,如 CPU 的频率、内部寄存器基地址、中断系统等。

(2) 确定当前处理器系统的 ppc _ md 结构,ppc _ md 结构确定当前 Linux PowerPC 的一系列钩子函数,设置 ppc _ md 函数的主要目的是为了进一步优化 Linux PowerPC 系统源代码的结构。

machine _ init 函数分别使用 early _ init _ dev _ tree 函数和 probe _ machine 函数,实现以上两种功能。machine _ init 函数有两个输入参数 dt _ ptr 和 phys,其中 dt _ ptr 指向 dtb 的物理地址,而 phy 指向 Linux 内核所在的物理地址。本书编写时, E500 内核还不支持 OF 结构。因此本章涉及的许多源代码与 E500 内核无关。

```
void __init machine_init(unsigned long dt_ptr, unsigned long phys)
{
...
  early_init_devtree(__va(dt_ptr));

  probe_machine();
...
}
```

1. early _ init _ dev _ tree 函数

early _ init _ dev _ tree 函数在 ./arch/powerpc/kernel/prom.c 文件中,该函数有一个输入参数 params,其值由 machine _ init 函数传入,用来存放 OF Tree 的有效地址,该函数没有返回

值。

```
void __init early_init_devtree(void *params)
{
...
  of_scan_flat_dt(early_init_dt_scan_chosen, NULL);
  lmb_init();
  of_scan_flat_dt(early_init_dt_scan_root, NULL);
  of_scan_flat_dt(early_init_dt_scan_memory, NULL);
```

early_init_devtree 函数首先调用 of_scan_flat_dt 函数对 dtb 的 chosen 节点进行分析。有关 chosen 节点的说明见本书的 8.1.1 节。之后该函数将根据 dtb 中的信息创建 LMB 结构，本书不对 LMB 结构进行说明。LMB 结构是 AIX 系统管理 eSeries 和 pSeries 服务器内存的一种策略，这一策略用于单处理器系统显得有些复杂。

```
/* Save command line for /proc/cmdline and then parse parameters */
strlcpy(saved_command_line, cmd_line, COMMAND_LINE_SIZE);
parse_early_param();
```

随后将 Boot Loader 引导 Linux 内核时使用的命令行参数存放到 saved_command_line 变量中，并使用 parse_early_param 函数分析这些命令行参数。

```
lmb_reserve(PHYSICAL_START, __pa(klimit) - PHYSICAL_START);
reserve_kdump_trampoline();
reserve_crashkernel();
early_reserve_mem();
```

上述程序将 Linux 内核使用的空间和 initrd 使用的空间在 LMB 中预留。

```
...
  of_scan_flat_dt(early_init_dt_scan_cpus, NULL);
} /* End early_init_devtree */
```

这段程序调用 early_init_dt_scan_cpus 函数，获得当前系统一共有多少个 CPU，同时设置哪一个 CPU 作为当前处理器系统使用的 BSP(Boot Strap Processor)。

2. probe_machine 函数

probe_machine 函数在 ./arch/powrepc/kernel/setup-common.c 文件中。该函数没有输入参数，也没有返回值。该函数的实现流程十分简单。

```
void probe_machine(void)
{
  extern struct machdep_calls __machine_desc_start;
  extern struct machdep_calls __machine_desc_end;
```

probe_machine 函数首先引用外部变量__machine_desc_start 和__machine_desc_end，这两个变量在 vmlinux.lds.S 文件中定义，Linux 系统编译完成后这两个变量被初始化。

```
DBG("Probing machine type ... \n");

for (machine_id = &__machine_desc_start;
    machine_id < &__machine_desc_end;
    machine_id++ ) {
    DBG(" %s ...", machine_id->name);
    memcpy(&ppc_md, machine_id, sizeof(struct machdep_calls));
    if (ppc_md.probe()) {
      DBG(" match ! \n");
      break;
    }
    DBG("\n");
}
```

上述程序以 machdep_calls 为单位扫描__machine_desc_start 到__machine_desc_end 之内的内存空间,并将每一个单位赋值到 ppc_md 结构中,如果相应的 ppc_md.probe 函数成功返回,则表示成功地找到了当前处理器系统所需要的 ppc_md 结构。Linux PowerPC 首先使用宏 define_machine,将基于各类 PowerPC 内核的 ppc_md 结构加入到从__machine_desc_start 到__machine_desc_end 之内的空间中。

宏 defin_machine 在./include/asm-powerpc/machdep.h 文件中,其源代码如下所示:

```
#define define_machine(name)                    \
  extern struct machdep_calls mach_##name;      \
  EXPORT_SYMBOL(mach_##name);                   \
  struct machdep_calls mach_##name __machine_desc =
```

该宏 machdep_calls 类型的数据加入到.machine.desc 段中。在一个处理器系统中,只能定义一种 machdep_calls 类型的数据。在 MPC8541 CDS 系统中,使用宏 define_machine 对 mpc85xx_cds 进行初始化。

```
define_machine(mpc85xx_cds) {
  .name       = "MPC85xx CDS",
  .probe        = mpc85xx_cds_probe,
  .setup_arch   = mpc85xx_cds_setup_arch,
  .init_IRQ = mpc85xx_cds_pic_init,
  .show_cpuinfo = mpc85xx_cds_show_cpuinfo,
  .get_irq = mpic_get_irq,
  .restart = mpc85xx_restart,
  .calibrate_decr = generic_calibrate_decr,
  .progress = udbg_progress,
};
```

Linux PowerPC 采用.machine.desc 类型对 ppc_md 结构进行赋值的主要优点是可以进一步将 Linux PowerPC 中与处理器无关的代码独立出来,当用户增加一个新的处理器系统时,

仅需要修改与自己系统有关的代码，而不必对 Linux PowerPC 的源代码进行大规模修改。

以 MPC85XX _ CDS 系统为例，系统程序员将此处理器系统添加到 Linux PowerPC 时，只需要在 ./arch/powerpc/platforms 目录中，添加相应的目录及一些与 MPC85XX _ CDS 系统相关的源代码即可，而不需要对其他源代码进行修改。保持 Linux 系统各个部分源代码的独立是 Linux 系统的开发维护者努力的目标。

8.2.6 MMU _ init 函数

MMU _ init 函数的主要作用是为 Linux PowerPC 建立存储器虚实映射，初始化 Linux 系统中的 PTE 表和 MMU 的一些设置。MMU _ init 函数没有输入参数也没有返回值，该函数的执行流程如下所示：

```
void __init MMU_init(void)
{
...
    MMU_setup();

    if (lmb.memory.cnt > 1) {
      lmb.memory.cnt = 1;
      lmb_analyze();
      printk(KERN_WARNING "Only using first contiguous memory region");
    }

    total_memory = lmb_end_of_DRAM();
    total_lowmem = total_memory;
```

- 这段程序首先执行 MMU _ setup 函数，分析 Boot Loader 程序引导 Linux PowerPC 时使用的命令行参数，有些应用可以在命令行参数中，对 Linux PowerPC 的存储器系统进行设置，MMU _ setup 函数仅能对"nobats"和"noltlbs"参数进行分析，该函数的源代码在 ./arch/powerpc/mm/init _ 32.c 文件中。
- 判断当前处理器系统中一共有多少个存储器区域，Linux PowerPC 内核只使用第一段物理地址连续的内存空间。
- 从 LMB 系统中获得当前处理器系统第一段物理地址连续的内存空间大小，并将此数值赋值到全局变量 total _ memory 和 total _ lowmem 中。

```
#ifdef CONFIG_FSL_BOOKE
/* Freescale Book-E parts expect lowmem to be mapped by fixed TLB
 * entries, so we need to adjust lowmem to match the amount we can map
 * in the fixed entries */
 adjust_total_lowmem();
#endif /* CONFIG_FSL_BOOKE */
```

如果当前处理器系统基于 E500 内核，Linux PowerPC 将会执行 adjust _ total _ lowmem 函数，该函数在 ./arch/powerpc/mm 目录的 fsl _ booke _ mmu.c 文件中。

基于 E500 内核的 PowerPC 处理器要求 Linux PowerPC 使用 TLB1 的 Entry 对内核地址空间进行虚实地址转换。因为与使用 TLB0 相比,使用 TLB1 实现虚实地址转换时,不需要使用存放在内存中的 PTE 表,因此效率较高。

Linux PowerPC 最多使用 TLB1 中的前 3 个 Entry 进行虚实地址映射,对于 E500 V1 内核,最多只能映射 768MB 地址空间。如果一个基于 32 位 PowerPC 处理器系统的 Linux 内核实用的内存超过 768 MB 时,需要使用 HIMEME 管理这段内存,或者不使用这段内存空间。Linux PowerPC 设置了宏 MAX_LOW_MEM,表示低端内存的最大值,对于基于 E500 V1 内核的处理器该值为 0x30000000,即 768 MB。

adjust_total_lowmem 函数将当前处理器系统中的内存分为三个部分。如果当前处理器系统的内存空间大于 768 MB,则只处理这 768 MB 之内的数据空间。然后将这段空间以 256 MB 为单位,分别存放到全局变量__cam0,__cam1,__cam2 中。如果当前处理器系统的内存大小为 256 MB,则__cam0 = 256 M,__cam1 = 0,__cam2 = 0;如果当前处理器系统内存大小为 760 MB,则__cam0 = 256 M,__cam1 = 256M,__cam2 = 248 M。

```
if (total_lowmem > __max_low_memory) {
  total_lowmem = __max_low_memory;
  total_memory = total_lowmem;
  lmb_enforce_memory_limit(total_lowmem);
  lmb_analyze();
}
```

__max_low_memory 是一个全局变量,其值在 ./arch/powerpc/mm/init_32.c 文件中。对于 MPC8541CDS 系统,该值为 0x3000-0000,即为 768 MB。在以上程序中,如果当前处理器系统的内存空间大于 768 MB,则将 total_lowmem,total_memory 的值置为 768 MB,并重新对 LMB 进行整理。如果 Linux 系统使能了 CONFIG_HIMEM 参数,这里的处理略有不同,本书对此不做详细介绍。有兴趣的读者,可以自行阅读这些源代码。

```
/* Initialize the MMU hardware */
MMU_init_hw();
```

MMU_init_hw 函数在 E500 内核中的实现较为简单,因为 E500 内核不支持 HPTE 表。对于 E500 内核,该函数的主要功能是刷新 E500 内核的指令 Cache。Boot Loader 程序将 Linux 内核以数据的形式复制到内存中,指令 Cache 对这个复制工作一无所知,因此必须对 E500 内核的指令 Cache 进行刷新,以保证系统指令 Cache 与程序的一致性。

MMU_init_hw 函数的实现流程如图 8-4 所示。

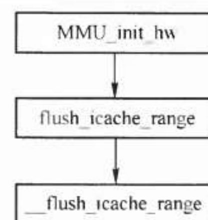


图 8-4 MMU_init_hw 函数执行流程

至此,Linux 系统完成了对物理内存的检查、修正与整理,随后将使用 mapin_ram 函数对 Linux 内核程序使用的物理地址空间进行虚实映射。

```
/* Map in all of RAM starting at KERNELBASE */
mapin_ram();
```

mapin _ ram 函数在 ./arch/powerpc/mm/pgtable _ 32.c 文件中,该函数没有输入参数也没有返回值。对于基于 E500 内核的处理器,将使用 TLB1 的前三个 Entry 对 Linux 内核程序使用的物理地址空间进行虚实映射。mapin _ ram 函数首先调用 mmu _ mapin _ ram 函数,使用 TLB1 的前三个 Entry 对 Linux 内核使用的物理地址空间进行虚实映射。mmu _ mapin _ ram 函数执行完毕后,mapin _ ram 函数将调用 map _ page 函数,使用 TLB0 映射剩余的内存空间。

```
...
    /* Map in I/O resources */
    if (ppc_md.progress)
        ppc_md.progress("MMU:setio", 0x302);
    if (ppc_md.setup_io_mappings)
        ppc_md.setup_io_mappings();
...
} /* End MMU_init */
```

Linux PowerPC 不建议用户使用 io _ block _ mapping 函数进行段式虚实地址转换,但是 Linux PowerPC 依然为用户提供了进行此操作的函数。如果用户需要使用该函数进行虚实地址的转换,首先需要使用 8.2.5 节提到的宏 define _ machine 将 setup _ io _ mappings 参数进行初始化,然后使用 setup _ io _ mappings 函数,调用相应的 io _ block _ mapping 函数完成虚实地址的转换。

提醒读者注意,io _ block _ mapping 函数使用的虚拟地址空间需要在 VM 空间之内,系统程序员需要精确计算出所有 io _ block _ mapping 函数使用的边界,以避免 Linux 系统虚拟地址空间的浪费。对于 32 位处理器,Linux 系统的 1GB 内核空间已经显得捉襟见肘。Linux 系统引导程序从 MMU _ init 函数返回时,将调用 start _ kernel 函数。

```
/* Let's move on */
lis r4,start_kernel@h
ori  r4,r4,start_kernel@l
lis r3,MSR_KERNEL@h
ori  r3,r3,MSR_KERNEL@l
mtspr  SPRN_SRR0,r4
mtspr  SPRN_SRR1,r3
rfi     /* change context and jump to start_kernel */
```

这段程序使用 rfi 指令跳转到 start _ kernel 函数开始执行。

8.3 Linux 内核的二次引导

Linux 内核的一次引导所完成的主要工作与当前处理器系统密切相关,包括对一些底层硬件进行最基本的初始化操作。而二次引导中的主要工作与处理器类型基本无关。

Linux 系统在进入 start _ kernl 函数运行之前,已经为该函数的执行建立了一个基本环境和必要的准备,并由二次引导程序完成对 Linux 内核的初始化。与一次引导程序相同,二次引

导程序也是被进程 0 调用的。Linux 内核的二次引导程序基本上是一个顺序执行的流水账。本书将按照程序的执行顺序对一些重要的函数进行说明。

8.3.1 start _ kernel 函数

start _ kernel 函数在 ./init/main.c 文件中。其源代码如下所示：

```
asmlinkage void __ init start _ kernel(void)
{
  char * command _ line;
  extern struct kernel _ param __ start ___ param[ ], __ stop ___ param[ ];
...
  local _ irq _ disable();
  early _ boot _ irqs _ off();
...
  lock _ kernel();
  boot _ cpu _ init();
  page _ address _ init();
  printk(KERN _ NOTICE);
```

- start _ kernel 函数首先调用 lock _ kernel 函数将 Linux 内核锁定。Linux 2.4 的初期版本使用 lock _ kernel 函数，实现内核抢占和 SMP 对 Linux 内核的互斥。lock _ kernel 函数使用 BKL(Big Kernel Lock)将内核锁定，在该锁之后执行的所有 Linux 内核代码都不能被其他处理器强占，直到使用 unlock _ kernel 函数释放 BKL。BKL 与 Semaphore 和 Spin Lock 有所不同，BKL 不是对临界数据的访问进行保护，而是对 Linux 内核进行保护，BKL 的效率非常低。
- 调用 early _ boot _ irqs _ off 函数将变量 early _ boot _ irqs _ enabled 置为 0，表示当前系统禁止外部中断。
- Linux PowerPC 在执行完 lock _ kernel 函数之前还调用了 local _ irq _ disable 函数，这使得 Linux 内核在引导过程中不会被外部中断干扰，也不可能被其他进程抢占。
- start _ kernel 函数调用 boot _ cpu _ init 函数确定 SMP 系统中的 BSP，即进程 0 使用的 CPU。
- 调用 page _ address _ init 函数，将 HIMEM 使用的空间组织在一起，page _ address _ init 函数只有在 Linux 系统使能 HIMEM 后才有意义。
- 调用 printk(linux _ banner)函数将 Linux _ banner 变量输出到控制台。Linux _ banner 变量在 ./init/version.c 文件中。

```
setup _ arch( &command _ line);
```

setup _ arch 函数是 start _ kernel 中执行的重要函数。该函数的主要作用是对内存系统进行基本初始化，之后调用 ppc _ md 结构的 setup _ arch 函数对当前处理器系统进行一些基本的初始化，该函数在 ./arch/powerpc/kernel/setup _ 32.c 文件中。

```
/* Warning, IO base is not yet inited */
void __init setup_arch(char **cmdline_p)
{
  *cmdline_p = cmd_line;

  /* so udelay does something sensible, assume <= 1000 bogomips */
  loops_per_jiffy = 500000000 / HZ;

  unflatten_device_tree();
  check_for_initrd();

  if (ppc_md.init_early)
    ppc_md.init_early();

  find_legacy_serial_ports();

  smp_setup_cpu_maps();

  /* Register early console */
  register_early_udbg_console();

  xmon_setup();
```

在上述代码中，setup_arch 函数首先设置 loops_per_jiffy 参数，然后对 OF tree 和 initrd 进行检查，最后对串行端口和监视口进行检查和设置。在 Linux 系统中，串行端口和监视口经常被用作调试端口，其中串行端口最为常用。

```
...
if (cpu_has_feature(CPU_FTR_SPLIT_ID_CACHE)) {
  dcache_bsize = cur_cpu_spec->dcache_bsize;
  icache_bsize = cur_cpu_spec->icache_bsize;
  ucache_bsize = 0;
} else
  ucache_bsize = dcache_bsize = icache_bsize
    = cur_cpu_spec->dcache_bsize;
/* reboot on panic */
panic_timeout = 180;

init_mm.start_code = PAGE_OFFSET;
init_mm.end_code = (unsigned long) _etext;
init_mm.end_data = (unsigned long) _edata;
init_mm.brk = klimit;
```

这段程序主要完成以下几个功能：

- 设置指令与数据 Cache 的行长度。其中 dcache_bsize 和 icahe_bsize 分别表示 L1 数据 Cache 和指令 Cache 的大小。当处理器将指令 Cache 和数据 Cache 进行统一编址时，u-

cache _ bsize 变量用来 Cache 的总大小。在基于 E500 内核的 Linux PowerPC 中,dcache _ bsize 和 icache _ bsize 都为 32,表示 L1 数据和指令 Cache 的大小为 32KB。而 ucache _ bsize 变量为 0。

- 随后将 panic _ timeout 变量设置为 180。Linux 系统进入 panic 时,180 秒钟后将重新启动。
- 这段程序最后设置进程 0 的 mm _ struct 结构。

```
/* set up the bootmem stuff with available memory */
do_init_bootmem();
if ( ppc_md.progress ) ppc_md.progress("setup_arch: bootmem", 0x3eab);
```

这段程序对 Boot Memory 进行初始化。该函数的详细描述见本书的 7.2.4 节。

```
ppc_md.setup_arch();
if ( ppc_md.progress ) ppc_md.progress("arch: exit", 0x3eab);
```

调用 ppc _ md.setup _ arch 函数对具体的处理器系统进行初始化。不同的处理器系统使用不同的 ppc _ md.setup _ arch 函数,ppc _ md.setup _ arch 函数在 probe _ machine 函数中设置。对于 MPC8541CDS 系统,ppc _ md.setup _ arch 函数等效于 mpc85xx _ cds _ setup _ arch 函数。

mpc85xx _ cds _ setup _ arch 函数在 ./arch/powerpc/platforms/85xx/mpc85xx _ cds.c 文件中,该函数主要完成以下功能:

(1) 根据当前处理器的频率,重新配置 loops _ per _ jiffy 参数。

(2) 添加 PCI 总线控制器。

(3) 设置 ROOT _ DEV 参数,确定根文件系统。

```
    paging_init();
} /* End setup_arch */
```

setup _ arch 函数最后调用 paging _ init 函数,对当前处理器系统的 Linux 内核的有效地址空间进行全面的初始化,paging _ init 函数的详细说明见本书的 7.2.2 节。start _ kernel 函数从 setup _ arch 函数返回后将执行以下代码:

```
unwind_setup();
setup_per_cpu_areas();
smp_prepare_boot_cpu(); /* arch-specific boot-cpu hooks */
```

- 在 Linux PowerPC 中,unwind _ setup 函数是一个空函数。
- 调用 setup _ per _ cpu _ areas 函数。如果处理器不使用 Linux SMP,setup _ per _ cpu _ areas 函数是一个空函数,否则 setup _ per _ cpu _ areas 函数将初始化每个 CPU 的 .data.percpu 数据段,之后调用 alloc _ bootmem 函数为每个 CPU 确定内存的大小。在 Linux SMP 中,每个 CPU 都有自己的专用数据段,该专用数据段为 .data.percpu,该数据段在 ./arch/powerpc/kernel/vmlinux.lds.S 文件中定义。Linux PowerPC 使用宏 per _ cpu 访问这些属于不同 CPU 的专用数据,使用宏 DEFINE _ PER _ CPU 将数据放到 CPU 的专用数据段 .data.percpu 中。

- 调用 smp _ prepare _ boot _ cpu 函数,设置当前 BSP 有效,即当前 BSP 使用的 CPU 处于活动状态,然后设置 current _ set 数组的 boot _ cpuid 为当前进程描述符。smp _ prepare _ boot _ cpu 函数在 ./arch/powerpc/kernel/smp.c 文件中。

```
/*
 * Set up the scheduler prior starting any interrupts (such as the
 * timer interrupt). Full topology setup happens at smp_init()
 * time - but meanwhile we still have a functioning scheduler.
 */
sched_init();
```

这段程序调用 sched _ init 函数初始化进程运行队列 rq。在 Linux SMP 系统中,每一个 CPU 都有自己的进程运行队列。

```
preempt_disable();
build_all_zonelists();
page_alloc_init();
```

这段代码首先调用 preempt _ disable 函数禁止抢占。之后调用 build _ all _ zonelists 函数。该函数调用__ build _ all _ zonelists 函数初始化每个 CPU 的存储器节点,并根据 Linux PowerPC 提供的数据区域,如 ZONE _ DMA 和 ZONE _ HIMEM,划分内存空间。这部分初始化也被称为存储器节点的初始化。

最后调用 page _ alloc _ init 函数,为系统设置一个 page _ alloc _ cpu _ notify 回调函数。该函数用来实现 CPU 的关闭与使能。一个 MPP 结构的处理器系统或者大型服务器含有几十到几千个 CPU,这些处理器系统有时需要临时关闭或者打开某些 CPU,此时 Linux 系统需要调用 page _ alloc _ cpu _ notify 函数增加或者移出某些 CPU。

```
printk(KERN_NOTICE "Kernel command line: %s\n", saved_command_line);
parse_early_param();
parse_args("Booting kernel", command_line, __start___param,
    __stop___param - __start___param,
    &unknown_bootoption);
```

这段函数使用 parse _ early _ param 和 parse _ args 函数解析 Boot Loader 传递给 Linux 内核的一些参数。

```
sort_main_extable();
trap_init();
rcu_init();
```

这段函数首先使用 sort _ main _ extable 函数,对异常调用表进行重新排序。Linux 系统将异常调用表存放在__ ex _ table 段中。对于 Linux PowerPC,trap _ init 是一个空函数,该函数对 Linux Pentium 有意义。rcu _ init 函数初始化 Linux PowerPC 的 RCU 部件。本书由于篇幅问题无法深入讨论 Linux 系统中的 RCU。

```
init_IRQ();
pidhash_init();
init_timers();
hrtimers_init();
softirq_init();
```

- 这段程序调用 init_IRQ 函数对中断系统进行初始化，init_IRQ 函数的详细说明见本书的 6.2.3 节。
- 调用 pidhash_init 函数对进程 PID 描述符的 hash 表进行初始化。PID 描述符的详细说明见本书的 5.1.1 节。
- 调用 init_timers 函数初始化 Linux 系统使用的定时器。
- 调用 hrtimes_init 函数初始化高精度(high resolution)定时器，与 Linux 系统提供的标准定时器相比，该定时器的精度更高。
- 调用 softirq_init 函数对 Linux 系统的软件中断进行初始化，该函数的详细描述见本书的 6.4.4 节。

```
timekeeping_init();
time_init();
profile_init();
if (! irqs_disabled())
  printk("start_kernel(): bug: interrupts were enabled early\n");
early_boot_irqs_on();
local_irq_enable();
```

这段程序的执行流程如下：

- 调用 timekeeping_init 函数初始化 Linux 系统的外部 RTC，即 timekeeping 器件。
- 调用 time_init 函数初始化 Linux 系统的定时器。Linux PowerPC 可以使用外部 RTC 作为系统的定时器也可以使用 TB 寄存器。
- 调用 profile_init 函数初始化 Linux 系统的 profile 功能。本书在 5.4.3 节中简单介绍了一些有关 profile 的知识。
- 调用 early_boot_irqs_on 函数将全局变量 early_boot_irqs_enabled 置为 1 表示当前系统使能了外部中断。
- 调用 local_irq_enable 函数使能外部中断。

```
/*
 * HACK ALERT! This is early. We're enabling the console before
 * we've done PCI setups etc, and console_init() must be aware of
 * this. But we do want output early, in case something goes wrong.
 */
console_init();
```

调用 console_init 函数初始化控制器 console，console 是 Linux 系统用于输出一些监控信息的设备，Linux 系统一般使用串口或者监视器作为 console 设备。

```
...
    vfs_caches_init_early();
    cpuset_init_early();
    mem_init();
    kmem_cache_init();
    setup_per_cpu_pageset();
```

- 初始化 VFS 系统使用的 dentry 和 inode 专用 SLAB 描述符。Linux 系统使用 dentry 结构保存有关文件目录的信息，使用 inode 结构保存有关文件的信息。
- 调用 mem_init 函数初始化所有页表描述符，并初始化 Buddy System。本书在 7.2.3 节中简单介绍了 mem_init 函数。
- 调用 kmem_cache_init 函数初始化 Linux 系统的 Slab 分配器。本书在 7.3.2 节中详细介绍了 kmem_cache_init 函数的详细实现。
- 初始化当前 CPU，即 BSP 的每一个数据区域，如 ZONE_DMA 和 ZONE_HIMEM 使用的 per_cpu_pageset 缓冲，有关 per_cpu_pageset 缓冲的详细说明见本书的 7.2.3 节。

```
numa_policy_init();
if (late_time_init)
  late_time_init();
calibrate_delay();
pidmap_init();
pgtable_cache_init();
```

- 调用 numa_policy_init 函数创建 numa_policy 和 shared_policy_node 使用的专用 Cache 描述符。
- 调用 calibrate_delay 函数。calibrate_delay 函数可以计算出，一个 CPU 在一秒钟内执行了多少次"一个极短的循环"，并根据计算出来的值得到 BogoMIPS 值，其中 Bogo 是 Bogus(伪)的意思。通过 BogoMIPS 的值，可以估算出当前处理器的性能。该程序由 Linus Torvalds 编写，BogoMIPS 可以衡量在 Linux 系统中，处理器的运算速度。Linux 系统用于估算当前处理器运算速度的算法比较简单，BogoMIPS 的值可以部分反映当前处理器的运行速度。当前处理器的 BogoMIPS，在/proc/cpuinfo 文件的最后一行中。
- 调用 pidmap_init 函数，建立 pid 结构使用的专用 Cache 描述符表 pid_cachep。
- 对于 64 位的 PowerPC 处理器，pgtable_cache_init 函数建立 PTE 表和 PMD 表使用的专用 Cache 描述符；32 位的 PowerPC 处理器使用 Buddy System 创建 PTE 表和 PMD 表。因此对于 32 位 Linux 系统，该函数为空。

```
...
    fork_init(num_physpages);
    proc_caches_init();
    buffer_init();
    unnamed_dev_init();
    key_init();
```

```
    security_init();
    vfs_caches_init(num_physpages);
    radix_tree_init();
    signals_init();
```

- 调用 fork_init 函数创建 task_struct 结构使用的专用 Cache 描述符 task_struct_cachep;确定当前 Linux 系统所能容纳的最大进程数 max_threads。
- 调用 proc_caches_init 函数使用的专用 Cache 描述符 sighand_cachep、signal_cachep、files_cachep、fs_cachep、vm_area_cachep 和 mm_cachep。
- 调用 buffer_init 函数创建 buffer_head 结构的专用 Cache 描述符 bh_cachep,Linux 的文件系统使用 buffer_head 结构保存来自外部设备的数据。
- 调用 key_init 函数和 security_init 函数初始化与信息安全有关的数据结构。
- 调用 vfs_caches_init 函数创建专用 Cache 描述符 names_cachep 和 filp_cachep,之后调用 dcache_init、inode_init、files_init、mnt_init、bdev_cache_init 和 chrdev_init 函数对 Linux 文件系统的各个子系统进行初始化。
- 调用 radix_tree_init 函数初始化 Linux 系统使用的 Radix Tree。Radix Tree 是一种搜索树,Linux 系统可以使用 Radix Tree 结构加快数据的检索速度。
- 调用 signals_init 函数初始化 Linux 系统的信号机制。

```
    /* rootfs populating might need page-writeback */
    page_writeback_init();

#ifdef CONFIG_PROC_FS
    proc_root_init();
#endif
```

- 调用 page_writeback_init 函数初始化文件系统的回调函数。在 Linux 文件系统中有许多“脏”页,这些在内存中的“脏”页不会立即与文件系统进行同步。只有在内存中的“脏”页到达一定的数量时,Linux 系统才会使用回调函数将这些“脏”页与文件系统同步。
- 调用 proc_root_init 函数创建 proc 根文件系统。

```
...
    acpi_early_init(); /* before LAPIC and SMP init */
    /* Do the rest non-__init'ed, we're now alive */
    rest_init();
} /* End start_kernel
```

调用 acpi_early_init 函数初始化 Linux 系统的高级配置与电源接口 ACPI(Advanced Configuration and Power Interface)。ACPI 是英特尔、微软和东芝共同开发的一种电源管理标准,可实现以下功能:

- 用户设置外部设备开关的时间。
- 指定计算机在低电压的情况下进入低功耗状态,以保证重要的应用程序运行。
- 操作系统可以在应用程序对效率要求不高的情况下降低时钟频率。

● 操作系统根据外设和主板的具体需求分配能源。
● 在无人使用计算机时使计算机进入休眠状态,但保证打开一些基本通信设备。
● 管理插即用设备。

acpi _ early _ init 函数返回后将执行 rest _ init 函数,rest _ init 函数没有输入参数也没有返回值,该函数的源代码如下所示:

```
static void noinline rest_init(void)
    __releases(kernel_lock)
{
    kernel_thread(init, NULL, CLONE_FS | CLONE_SIGHAND);
    numa_default_policy();
    unlock_kernel();

    /*
     * The boot idle thread must execute schedule()
     * at least one to get things moving:
     */
     preempt_enable_no_resched();
     schedule();
     preempt_disable();

     /* Call into cpu_idle with preempt disabled */
     cpu_idle();
}
```

● rest _ init 函数使用 kernel _ thread 函数创建新的核心进程 init。此时在 start _ kernel 函数中使用的 BKL 并没有释放,而且内核还禁止抢占,因此核心进程 init 不会立即执行。
● 调用 unlock _ kernel 函数释放 start _ kernel 函数释放 BKL。
● 调用 preempt _ enable _ no _ resched 函数,允许内核抢占。
● 调用 schedule 函数切换当前进程。在执行此 schedule 函数之前,Linux 系统只有两个进程,即初始化进程 0 和核心进程 init,其中核心 init 进程刚刚被 rest _ init 函数创建。因此调用 schedule 函数后,核心进程 init 将会继续运行。
● 当 Linux 系统的进程运行队列中没有活动进程时,调度程序将执行进程 0。此时进程 0 将首先禁止内核抢占,然后执行 cpu _ idle 函数。注意在 idle 进程放弃 CPU 时将会重新使能内核抢占。cpu _ idle 函数在 ./arch/powerpc/kernel/idle.c 文件中,该函数包含 idle 进程的程序体,初始化进程 0 完成 Linux 系统的初始化后,将转化为 idle 进程。

8.3.2 核心进程 init

核心进程 init 主要有以下功能:
● 引导 SMP 处理器中的其他 AP。
● 调用 do _ basic _ setup 函数将 Linux 系统的其他模块初始化。
● 更换核心进程 init 为普通进程 init,之后进行 Linux 系统的二次引导,即对 Linux 系统应

用程序的引导。

```
static int init(void * unused)
{
  lock _ kernel();
  /*
   * init can run on any cpu.
   */
   set _ cpus _ allowed(current, CPU _ MASK _ ALL);

   init _ pid _ ns.child _ reaper = current;

   cad _ pid = task _ pid(current);

   smp _ prepare _ cpus(max _ cpus);
   do _ pre _ smp _ initcalls();

   smp _ init();
   sched _ init _ smp();

   cpuset _ init _ smp();
```

在这段程序中,最重要的函数是 smp _ init 函数,smp _ init 函数引导 SMP 系统中的 AP。该函数将调用 cpu _ up->_ cpu _ up->__ cpu _ up->kick _ cpu->__ secondary _ start 函数将 AP 逐个进行引导。在__ secondary _ start 函数中,AP 将调用 cpu _ set 函数将 CPU 加入到 cpu _ online _ map 变量中,用以向 BSP 通知,该 CPU 已经被激活。

需要提醒读者注意,在 AP 中,不再调用 start _ kernel 函数对 AP 使用的 Linux 子系统进行初始化,而是直接调用__ secondary _ start 函数对 AP 进行一些基本的初始化。在 SMP 处理器中,AP 与 BSP 共享同一个内存空间,因此在 Linux SMP 中只需要 BSP 对 Linux 子系统进行初始化,AP 将享受 BSP 的劳动成果。

```
do _ basic _ setup();
```

do _ basic _ setup 函数将调用 do _ initcalls 函数加载 Linux 系统中的各个模块,该函数最重要的源代码如下所示:

```
static void __ init do _ initcalls(void)
{
  initcall _ t * call;
  int count = preempt _ count();

for (call = __ initcall _ start; call < __ initcall _ end; call++ ) {
...
  result = ( * call)();
...
  }
} /* End do _ initcalls * /
```

上述程序将逐个执行在__ initcall _ start 和__ initcall _ end 之间的函数。__ initcall _ start 和__ initcall _ end 在 ./arch/powerpc/kernel/vmlinux.lds.S 文件中定义，Linux 系统将一些需要在系统启动时加载模块的地址存放在__ initcall _ start 和__ initcall _ end 之间，即 Linux 内核 ELF 文件的 .initcall.init 段中。

Linux 系统使用两种方式加载系统中的模块。

- 静态加载，即将所有模块的程序编译到 Linux 内核中，由 do _ initcalls 函数加载。
- 动态加载，即在 Linux 系统启动之后由 load _ module 函数进行加载。

本书重点介绍模块的静态加载方法。Linux 系统静态加载模块的方法十分简单，只需要调用 do _ initcalls 函数即可实现。但是程序员需要了解 Linux 系统如何将模块的程序地址放入 .initcall.init 段中。Linux 系统定义了一系列宏，将代码加入到__ initcall _ start 和__ initcall _ end 之间。

```
#define pure_initcall(fn)            __define_initcall("0",fn,1)
#define core_initcall(fn)            __define_initcall("1",fn,1)
#define core_initcall_sync(fn)       __define_initcall("1s",fn,1s)
#define postcore_initcall(fn)        __define_initcall("2",fn,2)
#define postcore_initcall_sync(fn)   __define_initcall("2s",fn,2s)
#define arch_initcall(fn)            __define_initcall("3",fn,3)
#define arch_initcall_sync(fn)       __define_initcall("3s",fn,3s)
#define subsys_initcall(fn)          __define_initcall("4",fn,4)
#define subsys_initcall_sync(fn)     __define_initcall("4s",fn,4s)
#define fs_initcall(fn)              __define_initcall("5",fn,5)
#define fs_initcall_sync(fn)         __define_initcall("5s",fn,5s)
#define rootfs_initcall(fn)          __define_initcall("rootfs",fn,rootfs)
#define device_initcall(fn)          __define_initcall("6",fn,6)
#define device_initcall_sync(fn)     __define_initcall("6s",fn,6s)
#define late_initcall(fn)            __define_initcall("7",fn,7)
#define late_initcall_sync(fn)       __define_initcall("7s",fn,7s)
```

上述代码使用宏__ define _ initcall 将 Linux 系统中需要静态加载的模块添加到 .initcall.init 段中。我们可以从以上程序发现在 Linux 系统中，模块分为 14 个等级，并使用 1～7，1s～7s 进行编号。其中，编号小的模块将放入 .initcall.init 段靠前的位置，这也保证了编号较小的模块将被率先执行。对此有兴趣的读者，可以阅读 Linux 源代码以了解 Linux 系统如何对这些模块进行分类。

在这些宏定义中，读者最熟悉的应该是宏 device _ initcall。编写过 Linux 系统设备驱动程序的读者应该使用过 module _ init 函数。如果 Linux 系统静态加载模块时，module _ init 函数等效于宏__ initcall，宏__ initcall 等效于宏 device _ initcall。

这些宏的代码在 ./include/linux/init.h 文件中。Linux 系统执行完 do _ basic _ setup 函数后，将继续执行以下代码：

```
...
free_initmem();
unlock_kernel();
```

- 调用 free_initmem 函数释放在__init_begin 和__init_end 之间的所有内存空间，即所有带有__init 前缀函数使用的内存空间。
- 调用 unlock_kernel 函数释放 BKL。

```
if (sys_open((const char __user *) "/dev/console", O_RDWR, 0) < 0)
    printk(KERN_WARNING "Warning: unable to open an initial console. \n");

(void) sys_dup(0);
(void) sys_dup(0);
```

这段程序使用/dev/console 作为 Linux 系统的标准输入，/dev/console 为 Linux 系统的文件描述符 0，之后使用 sys_dup 系统调用将文件描述符 0 复制到文件描述符 1 和 2 中。

在 Linux 系统中，文件描述符 0、1 和 2 分别与标准输入、标准输出和标准错误输出设备对应。因此在这段程序执行完毕后，系统的标准输入、标准输出和标准错误输出设备都被设置为/dev/console。

```
...
    run_init_process("/sbin/init");
    run_init_process("/etc/init");
    run_init_process("/bin/init");
    run_init_process("/bin/sh");

    panic("No init found. Try passing init= option to kernel.");
} /* End init */
```

init 函数最后将调用 run_init_process->kernel_execve->sys_execve 函数改变核心进程 init 的正文段，将核心进程 init 转换为用户进程 init。之后用户进程 init 将根据/etc/inittab 中提供的信息完成应用程序的初始化调用。本书将不介绍普通进程 init 的执行流程。

参考文献

[1] EREF: A Reference for Motorola Book E and the e500 Core, Rev. 2, by Freescale Semiconductor LTD.
[2] PowerPC™ e500 Core Family Reference Manual, Rev. 1, by Freescale Semiconductor LTD.
[3] PowerPC™ e500 Application Binary Interface User's Guide, Rev. 1.0, by Freescale Semiconductor LTD.
[4] PowerPC User Instruction Set Architecture Book I, Version 2.02, by IBM Corporation.
[5] PowerPC Virtual Environment Architecture Book II, Version 2.02, by IBM Corporation.
[6] PowerPC Operating Environment Architecture Book III, Version 2.02, by IBM Corporation.
[7] The Open Programmable Interrupt Controller (PIC) Register Interface Specification, Revision 1.2, Jointly by Advanced Micro Devices and Cyrix Corporation.
[8] Multiprocessor Interrupt Controller Data Book, by IBM Corporation.
[9] Storage in the PowerPC, by Janice M. Stone & Robert P. Fitzgerald, IBM T.J. Watson Research Center.
[10] Implementation of Atomic Primitives on Distributed Shared Memory Multiprocessors, by Maged M. Michael and Michael L. Scott.
[11] A New Approach for the Verification of Cache Coherence Protocols, by Fong Pong, Member, IEEE, and Michel Dubois, Senior Member, IEEE Computer Society.
[12] A High Performance Bus and Cache Controller for PowerPC™ Multiprocessing Systems, by Michael S. Allen, W. Kurt Lewchuk and John D. Coddington.
[13] Modeling and Verification of Cache Coherence Protocols, by Lubomir Ivanov & Ramakrishna Nunna.
[14] The Effects of Out-of-Order Execution on the Memory System, by Aamer Jaleel.
[15] Dynamic Dependency Analysis of Ordinary Programs, by Todd M. Austin & Gurindar S. Sohi.
[16] An Efficient Algorithm for Exploiting Multiple Arithmetic Units, by R. M. Tomasulo.
[17] The Design Space of Register Renaming Techniques, by Dezsö Sima.
[18] Designing the PowerPC 60X Bus, by Michael S. Allen & Michael Alexander & Chuck Wright & Joe Chang.
[19] The slab allocator: An object-caching kernel memory allocator, by Jeff Bonwick.
[20] Extending the slab allocator to many CPUs and arbitrary resources, by Jeff Bonwick and Jonathan Adams.
[21] Modern Operating Systems, 2nd Edition, by Andrew S. Tanenbaum.
[22] Understanding the Linux Virtual Memory Manager, by Mel Gorman.
[23] Linux Device Driver,3rd Edition, by Jonathan Corbet & Alessandro Rubini & Greg Kroah-Hartman.
[24] Unreliable Guide To Locking, by Paul Rusty Russell.
[25] Purely Functional Data Structures, by Chris Okasaki.
[26] Linux 2.6.20 源代码,http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.20.3.tar.bz2.
[27] Gcc 用户手册,http://gcc.gnu.org/onlinedocs.
[28] Linux Kernel Mail-list, linux-kernel@vger.kernel.org.

